Structured Programming

Suresh K. Basandra

Data processing managers are increasingly realising the importance of programming function in computer installations. A lot of effort in terms of time, money and manpower is involved in programming and systems analysis. In addition, much of the programming effort is directed towards program maintenance, i.e. changes in, or modifications to, existing programs. In such an environment, traditional programming comes into conflict with cost management objectives.

Traditional programming means programming generated as a form of personal art without adherence to established concepts and principles. The traditional way has been to view programming as a personal creation by an individual. It is quite possible for a programmer to write a clever program for solving a problem—the word 'clever' is often synonymous with 'complex' and 'obscure'. The difficulty with such a program is that it is hard to understand by persons other than the author and even the author may find it difficult to follow it after a long gap of time.

The recent past has seen a spectacular decrease in the hardware costs of computers. This trend is still continuing. As a result, cost of personnel relative to the hardware costs is increasing at a rapid rate. This calls for methods that can increase the personnel performance and thereby reduce the cost of program development. Quite often, the purpose for which a program is developed gets changed. In such cases, there are two alternatives: either the program should be declared obsolete and a new program developed in its place or the program should be modified incorporating the necessary changes.

The cost of personnel being on the increase, the question of discarding a program altogether does not arise. So, the program should be maintained by modifying it. This again calls for methods for program development so that the program is understood and modified easily. Programmers have, therefore, been attempting to find ways and means of writing correct programs which can be easily maintained, modified and debugged; for, such factors tend to become very important, especially in a rapidly changing business environment where similar programs have to be run frequently. This effort has led to the development of a programming discipline called structured programming.

As with all new concepts, there is considerable disagreement and misunderstanding as to what structured programming means. In fact, there is no standard definition of structured programming except that it is a disciplined approach to programming. In general, the term 'structured programming' has come to mean a collection of principles and practices that are directed towards developing correct programs which are easy to understand and maintain.

Some programmers insist that they have been using this technique for a long time. This may well be true, but the recent emergence of this technique has evolved from an attempt to formalise the process of designing programs in the same manner as the logic design has been formalised. The techniques used are, in general, not new but the formal basis definitely is.

Design considerations

The process of designing and writing a program can be sub-divided into a number of tasks such as:

(a) understanding the problem;
(b) producing an algorithm to solve the problem;
(c) coding the algorithm in a particular language;
(d) testing the resultant program; and
(e) iterating round the above tasks until the program is correct.

In conventional programming, tasks (a) and (b) are undertaken by the systems analyst and (c) and (d) by the programmer, with (e) being shared between both. This decomposition process can be thought of in a number of

ways but structured programming would suggest that it should be looked at as a top-down analysis, i.e. each stage is an elaboration of the previous stage with a greater degree of detail and complexity. This means that the problem is initially specified at a relatively low level of complexity and detail, and that the problem is gradually elaborated to produce the final program by stepwise refinement.

The nature of the difficulties involved in each step depends upon the problem and on the constraints on its solution, such as the resources available. The main difficulty in any problem-solving situation is to contain the complexity of the problem. It is this complexity which provides the intellectual challenge for programming. In structuring the solution, an attempt is made to simplify the complexity and to aid understanding. This approach is more likely to result in a correct solution.

Structured programming offers a number of benefits to its users. In this programming, each step is independent of the other steps, hence allowing separate checks at each step. Each step may be verified by checking the elaboration stage by stage and an error may, therefore, be detected in a systematic manner. At any one time, only a small amount of information has to be remembered and manipulated and the structure evolved is suitable for a rigorous proof of the correctness of the entire algorithm.

Objective and principles

The objective of structured programming is to provide methodologies so that:

(a) programs are developed quickly and with fewer mistakes;

(b) programs are read and understood easily; and

(c) a portion of the program can be modified without upsetting the functions of other portions.

In other words, the objective is to meet the challenge offered by the changing trends in data processing. The principles of structured programming are as follows:

(a) structuring of control flow;

(b) decomposing a program into 'modules' or 'partitions'; and

(c) top-down approach towards program designing.

These principles and the methodologies to implement them are outlined below.

Program structure

A computer program is a set of instructions to the computer. But while the instructions are to be executed by the computer, the language in which we communicate these instructions is meant to be comprehensible to humans. High-level languages like COBOL are intended for human use in their direct form and are intended for machine use only indirectly, through the process of compilation. Therefore, a basic principle of good programming is that the program be easily understood.

A program that can be understood can also be analysed and tested for correctness. Incorrect programs have plagued computing from its early days. Another basic principle of good programming then is to write correct programs. In a more theoretical vein, we talk of proof of program correctness as a reflection of efforts to prove in logical and unambiguous terms that a piece of program code is correct, i.e. it does what it is intended to do and nothing else. Theoretical progress has been made only in very limited contexts, and at present we cannot rely on a formal theory to guide us to produce correct programs. Instead, we rely on a number of rules and guides that are closer to commonsense than they are to a formal theory.

Every program has a structure. If the structure is well-defined, the program is correct and easy to understand and modify. On the other hand, a poorly structured program may have errors that are difficult to detect, may be hard to read and may be troublesome to modify. In cases where a change has to be made, it may become necessary to scrap the whole program and start afresh with a new one. In contrast, a structured program is characterised by clarity and simplicity in its logical flow structure. It reads like ordinary language from the beginning to the end, instead of branching from later paragraphs to earlier ones and back again. So, a simple, straight flow of logic is another principle of good programming.

Programming by itself is a complex task and the aim of structured programming is to reduce and control complexity as it is not possible to eliminate the complexity completely. Human intellect can comprehend only a limited amount of data at a given time. When human mind is faced with a complex task, it tries to manage by breaking it into a series of well connected smaller parts. Each smaller part or module, as it is called, is simple and comprehensible by itself so that within each module, all simplicity and clarity is achieved. So one of the characteristics of a well structured program is that it consists of a large number of small modular programs. Each modular program is self-contained and performs a single function. Each module can be tested and debugged independently.

This means that every module has only one input and one output point, though within the module there may be branches and loops. Of course, the programmer has to use his commonsense in dividing the complex program into a large number of smaller self-contained units or modules. Since each module has a specific independent function, it can be perfected separately without mixing it with the other parts of the program. This is indeed a simplification of the programming art.

How is a modular structure developed? In practice, a modular structure is developed by proceeding from the general to the specific, or by using what is widely known as the top-down approach. Starting with the barest specification of the problem, one goes about filling the details. Suppose we are to develop a pay-roll processing problem utilising the top-down approach. The most abstract statement of the task

is 'write pay-roll'. Then one can proceed to breakdown this general statement to more specific statements, such as:

Edit input data
Process against master file
Output pay-roll checks and other reports

Each of these broad functions can be subdivided into more specific ones, as for example, we can divide the first job still further as follows:

Check the valid employee number
Check time card data against job tickets
Check for valid department number, and so on.

The first program module which may be called the Read Module performs a certain function after making all checks as given by commands or statements within that module. The main point is that the execution of commands or statements within the module has nothing to do with statements in other modules and that is why we say each module can be independently developed, tested and debugged. In the overall structure of the program each module is taken as a single instruction.

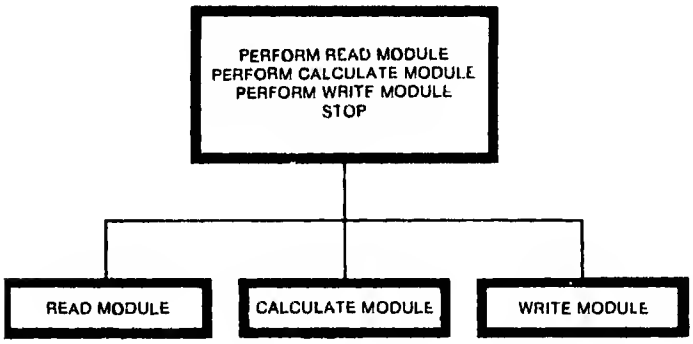

Fig. 1: Modular hierarchy diagram.

If a problem can be divided into, say, five modular functions, the main program can be considered to be made of five independent instructions, each instruction performing a module. A program organisation is, therefore, first divided into modular organisation and then each module is further divided into a statement organisation. Just as a government organisation performs the governing function by a hierarchy of officials at different levels, a complex programming task is achieved by dividing the job among different modules organised in a particular hierarchical structure. This is the top-down approach and this seems to work well in all human tasks. Fig. 1 shows what is called modular level or modular hierarchy diagram of the above problem.

It should be noted that no programming language is used when developing a programming structure. The thoughts are simply written in ordinary language with or without graphic aids such as arrows, boxes, brackets etc and the whole function is divided into a large number of simpler functions in a logical way. So, structured programming is not concerned with any particular language. It is concerned with programming in general.

Structuring of control flow

Except for the most trivial examples, programs are highly complex systems of logic. The ability to design loop operations and to take alternative courses of action depending on a condition can make a program very complex. This is because, quite often, loops are used within alternative paths or loops and alternative paths are used within loops or other alternative paths. The large size of a program and the interconnections between its various parts through the transfer of control also increase the program complexity.

The idea behind the structuring of control flow is to keep this complexity under control so that the design and understanding of the program logic become easier. It has been demonstrated that a structured program can be completely developed using three basic forms of program structure, viz. (i) sequence structure, (ii) decision or selection structure, and (iii) loop structure. One may encounter these as special terms which are written: SEQUENCE, IFTHENELSE, DOWHILE.

Sequence structure. The SEQUENCE structure indicates the sequential flow of program logic. Each block may stand for a statement; it may stand for a whole module or even a collection of modules. In a SEQUENCE structure the statements are executed one after the other in the order in which they are written. The flowchart of such a structure is shown in Fig. 2. The flow of control is such that it performs the block A first, then the block B, and so on, from top to bottom in the same direction. There is only one entry point

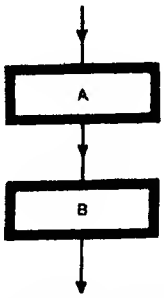

Fig. 2: Flowchart of the SEQUENCE structure.

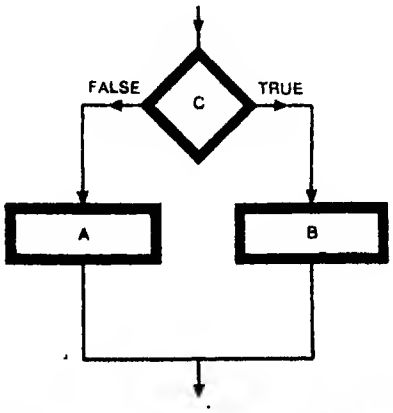

Fig. 3: Flowchart of the IFTHENELSE structure.

and one exit point for the structure and there is no complexity. The flow is in one direction and straightforward.

Decision structure. The IFTHENELSE structure indicates conditional program flow. The program takes one path or another, depending on whether a condition, often referred to as the predicate, is true or false. The program on entering the test box C in Fig. 3 makes a decision to go through either of the two paths A or B. After performing one of the two paths, the control returns to a single exit point. It can be seen that this structure also has one entry point and one exit point, even though there are multiple exit paths inside the structure.

Loop structure. The DOWHILE structure provides for a looping operation, i.e. repetitive execution of a program segment. A loop is not infinite if it is correct. The predicate tests for a condition. If the condition holds, we exit from the loop; if not, we execute the instruction. The flowchart of this structure is shown in Fig. 4. Whether the operations are to be repeated or stopped is decided in the decision box.

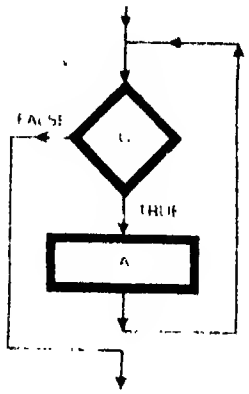

Fig. 4: Flowchart of the DOWHILE structure.

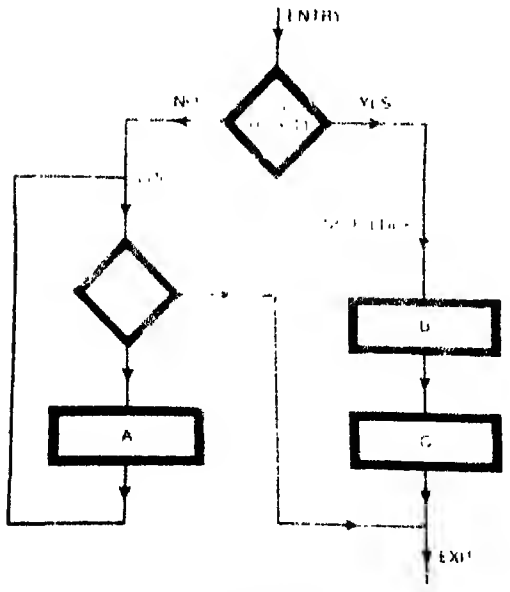

Fig. 5: Flowchart of the nested structure.

Any program can be written by using these three structures as the basic building blocks. In complex programs, one structure may be embedded in the other. For example, in Fig. 5, a nested structure is shown. When selection between the two available paths is made in the decision block C1, the control enters either a loop structure on the left or a sequence structure on the right. After performing anyone of the two paths, the control returns to a single exit point. In a similar manner, any number of structures can be embedded, provided the combination as a whole has one entry point and one exit point.

Modular programming

Modular programming is a programming strategy where a program is divided into a number of identifiable partitions or modules. Thus, a module in itself is a proper program structure. It has one entry and one exit, reads naturally from top to bottom, and may consist of any of the three basic program structures (SEQUENCE, IFTHENELSE, DOWHILE). The module is referenced by a name which should be as descriptive as possible. In COBOL that name is a paragraph or section name. A module should not exceed one page in length so that it is easily read. Finally, each module is executed by reference to it from another part of the program, usually through the PERFORM verb.

Fig. 6 depicts a general outline of structured programming logic. Note that there is a main logic module and a number of other dependent modules. The main logic part provides a summary of the program, and it is generally short enough so that the essence of the entire program can be understood in the context of the main module. Notice that module A, for instance, is executed under control of the main program module. If the name of module A is meaningful, say, Compute - total-deductions, we need not bother with the details of module A.

It may very well be the case that, as in this example, module A is sufficiently large or complex so that we need to break a module X out of it. Thus, module X is executed under the control of module A which in turn is executed under the control of the main module. This nesting of modules is permissible and desirable. We only need to keep in mind that as soon as the end of a module has been reached, we should turn to the module which issued the execute instruction. Thus, when module X has been executed, we return to instruction 2 of module A and, to the PERFORM MODULE B instruction of the main program module.

The program structure described above is adequate for most data processing. However, there are two conditions that restrict its use. Some tasks are common to more than one program. Repeating the same instructions in each such program is undesirable. The second case concerns large programming tasks. It is difficult for a team of programmers to work on the same program at the same time. It is easier to sub-divide the composite task into independent program

modules that can be written, compiled, and tested by individual programmers working independently of one another.

This latter objective is achieved by using program subroutines, or subprograms. A program subroutine is a program just like any other program, except that it is not executable by itself; it can only be executed under control of another program. It should be clear, however, that with or without the use of program subroutines, the COBOL programmer can create modular, well-structured programs.

```
Begin Main Program Module
Instruction 1
●
●
●
Instruction M
PERFORM Module A
PERFORM Module B
●
●
●
Instruction n
End Main Program Module
Begin Module A
Instruction 1
PERFORM Module X
●
●
●
Instruction n
End Module A

Begin Module B
  Instruction 1
  ●
  ●
  ●
  Instruction n
End Module B

Begin Module X
  Instruction 1
  ●
  ●
  ●
  Instruction n
End Module X
```

Fig. 6: A general outline of structured programming logic.

The advantage of modular programming depends on how effectively the modules are designed. Each module must be designed to accomplish a distinct function. This gives the program a rather high modification potential. For example, in a pay-roll program there may be a module that calculates the dearness allowance from the basic pay using certain rules. If these rules get changed on a later date, the modification will be limited to the said module and/or to its subordinate modules, if any. It has been seen that it is easier to replace a module by a modified one than to make corrections in one particular part of a conventional program without upsetting the functions of the other parts.

Top-down approach

The above description of modular programming presupposes a hierarchical structuring of modules. The process of designing a program consisting of a hierarchical structure of modules can be viewed in two ways: top-down, and bottom-up.

The top-down approach towards program design starts with the specification of the function to be performed by a program and then breaks it down into progressively subsidiary functions. The division of the function progresses with increasing levels of details. Each function at each level is ultimately realised in the form of a module.

In this approach, the calling module is always designed before the called module. At the time of designing a module, the broad functions to be performed by its immediate subordinate modules are assumed. The details of how a subordinate module can perform the specified functions are not considered until the subordinate module is taken up for design. Thus, the top-down approach represents a successive refinement of functions and this process of refinement is continued until the lowest modules can be designed without further analysis.

The top-down structure can be viewed as a tree structure, a typical example of which is shown in Fig. 7. Each box in this figure is a module. The topmost module denoted by 1 represents the program which can be called as main-line module or main-control module. In this case, the main-control module is divided into three subordinate modules denoted by 2, 3 and 4. The modules 2 and 4 require further divisions and in this process, the terminal modules are 5, 6, 7 and 8. The functions of these terminal modules are assumed to be simple enough to be easily programmed in the source language.

The bottom-up approach is the reverse of the top-down one. The process starts with the identification of a set of modules which are either available or to be constructed. An attempt is made to combine the modules to form modules of a higher level. This process of combining modules is continued until the program is realised. The basic drawback of the bottom-up approach is the assumption that the lowest level modules can be completely specified beforehand, which in reality is seldom possible. Thus in the bottom-up approach, quite often it is found that the final program obtained by combining the predetermined lowest level

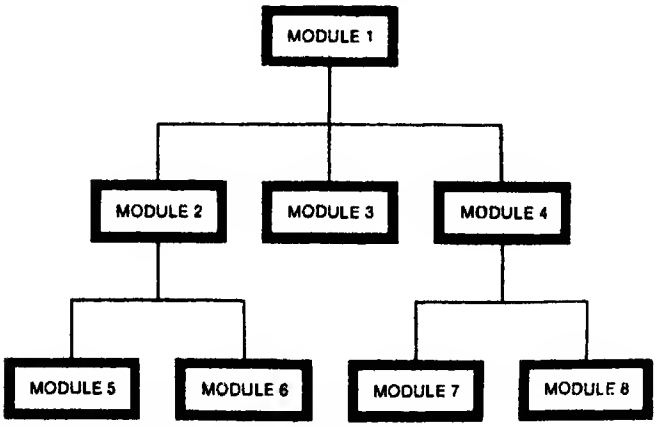

Fig. 7: A typical top-down structure.

modules does not meet all the requirements of the desired program.

No attempt is made here to compare the advantages and disadvantages of the two approaches. However, program development through top-down approach is widely accepted to be better than the bottom-up approach. The top-down approach has the following advantages:

1. It imitates the human tendency to solve a problem by outlining the broad concepts first and then subsequently going into the details.

2. The details of a module can be worked out with no (or minimum) change of the previously outlined concepts regarding its functions

3. The programmer never loses sight of the assumptions made at the previous levels. The development of modules can take place in parallel.

These advantages suggest that if the top-down approach is taken for program design, the programs can be developed easily and quickly, committing minimum of errors.

Constrained use of GO TO

The proponents of structured programming appear to be divided on the question of using the GO TO statement in structured programs. While some recommend the total avoidance of GO TOs in structured programs, others favour its constrained use. Without entering into this controversy, it may be stated that the GO TO statements need not be avoided just for the sake of avoiding them. Only when their use requires a compromise with the readability of the program, must they be eliminated.

Some general rules for the constrained use of GO TOs in COBOL programs are given below (It may be noted that each rule is more restrictive than the preceeding one.):

1. Use GO TO to transfer the control within a module without crossing its boundaries. In other words, a short-range GO TO statement whose object is either the name of the paragraph in which it appears or the name of any other paragraph in a series of paragraphs that constitutes a module is allowed according to this rule.

2. Use GO TO to transfer the control in the forward direction within a module. This rule is more restrictive than the previous one as it does not allow the use of the statement to transfer the control in the backward direction.

3. Use GO TO to transfer the control to the exit paragraph of a series of paragraphs constituting a module.

It is worthwhile to note that these rules ensure a localised flow of control so that the readability of the module is not likely to be seriously affected. Moreover, the module as a whole can still be viewed to have a single entry and single exit.

Programming considerations

How do the concepts of structured programming produce better programs? If we take the division of a problem into its programming solution, we see that the programming solution must contain a structure of 'boxes within boxes' in a pictorial representation where boxes within boxes represent the lower level elaboration at the upper level (Fig. 8). This means that we have a modular structure where modules can be tested independently.

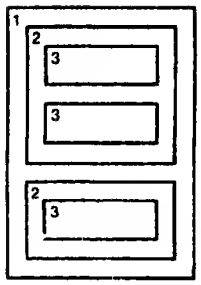

Fig. 8: Block structures (numbers represent the levels).

It should be pointed out here that we have a structure which looks, in some respects, similar to a flowchart which gives a pictorial representation of the program structure. The main difference is that the concepts in structured programming place restraints on the equivalent flowchart representation. The shortcomings of flowcharts are that they are too general and do not constrain the structure to be "well-formed". The main problem is the undisciplined use of the GO TO or JUMP instructions which may cause convoluted flowcharts. Here structured flowcharts may be used to good advantage.

Structured flowcharts

Traditionally, flowcharts have been used for two main purposes: to help the programmer develop the program logic, and to serve as documentation for a completed program. In recent years there has been some tendency to minimise the need or even the usefulness of flowcharts. A case can be made that if a programmer utilises principles of good program structure, the program is self-documenting and therefore a flowchart is superfluous. Further, there are those who argue that in an environment where program modifica-

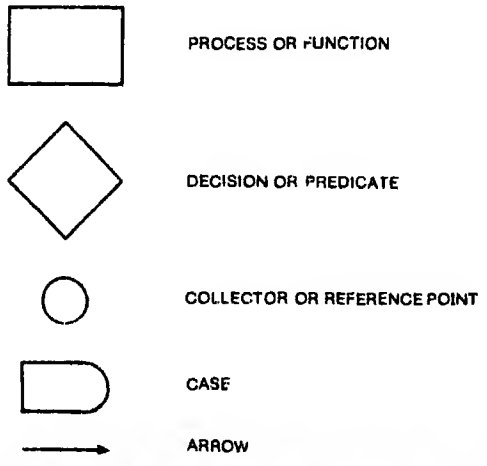

Fig. 9: IBM structured program flowcharting symbols.

tions are frequent, redrawing flowcharts becomes cumbersome. Instead of subscribing to the extreme view that flowcharts are useless, flowcharting can be viewed as a very useful tool for program development and documentation. But this programming aid should not be indiscriminately used.

Structured flowcharts are intended to focus on the program logic rather than on the specific tasks to be performed by the program. As such, structured flowcharts tend to do away with special symbols for input/output, and the like. While there is no standard set of symbols, Fig. 9 presents a set of flowcharting symbols adopted by IBM. A complete flowchart can be drawn using these symbols alone, although one may use standard flowcharting symbols as well. In any case, a structured flowchart is clear, simple and easy to understand. It has one clear beginning and one clear end and the advantage of utilising the five basic structures (SEQUENCE, IFTHENELSE, DOWHILE, DOUNTIL, and CASE).

Structured programming is a good approach for writing correct programs. The strategy behind this technique is: when the foe is formidable, divide and conquer. Such a disciplined approach to programming tends to minimise errors and maximise output, which is the concern of all good programmers.

□