2 Programming Methodology

2.1 GENERAL CONCEPTS

A **program** is a sequence of instructions written in a programming language. There are various programming languages, each having its own advantages for program development. Generally every program takes an input, manipulates it and provides an output as shown below :

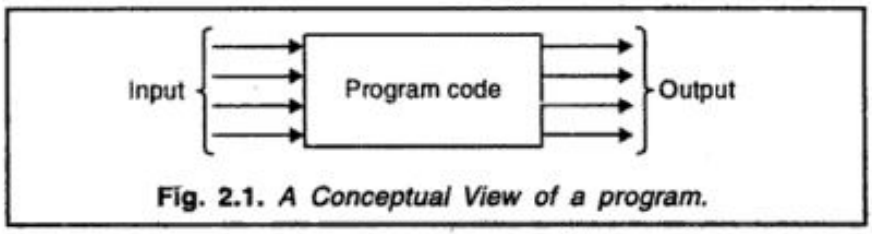

Fig. 2.1. *A Conceptual View of a program.*

For better designing of a program, a systematic planning must be done. Planning makes a program more **efficient** and more **effective.** A programmer should use planning tools before coding a program. By doing so, all the instructions are properly interrelated in the program code and the logical errors are minimized.

There are various planning tools for mapping the program logic, such as **flowcharts, pseudocode** and **hierarchy charts** etc. A program that does the desired work and achieves the goal is called an effective program whereas the program that does the work at a faster rate is called an efficient program.

The software designing includes mainly two things--*program structure* and *program representation.* The program structure means how a program should be. The program structure is finalised using top-down approach or any other popular approach. The program structure is obtained by joining the subprograms. Each subprogram represents a logical subtask.

The program representation means its presentation style so that it is easily readable and presentable. A user friendly program (which is easy to understand) can be easily debugged and modified, if need arises. So the programming style should be easily understood by everyone to minimize the wastage of time, efforts and cost.

Change is a way of life, so is the case with software. The modification should be easily possible with minimum efforts to suit the current needs of the organization. This modification process is known as **program maintenance.**

2.2 CHARACTERISTICS OF A GOOD PROGRAM

The different aspects of evaluating a program are : efficiency, flexibility, reliability, portability and robustness etc. These characteristics are given below :

(*i*) **Efficiency.** It is of three types : programmer effort, execution time and memory space utilization. The high level languages are used for programmer efficiency. But, a program written in machine language or assembly language is quite compact and takes less machine time, and memory space. So depending on the requirement, a compromise between programmer's effort and execution time can be made.

(*ii*) **Flexibility.** A program that can serve many purposes is called a flexible program. For example, CAD (Computer Aided Design) software are used for different purposes such as : Engineering drafting, printed circuit board layout and design, architectural design. CAD can also be used in graphs and reports presentation.

(*iii*) **Reliability.** It is the ability of a program to work its intended function accurately even if there are temporary or permanent changes in the computer system. Programs having such ability are known as reliable.

(*iv*) **Portability.** It is desirable that a program written on a certain type of computer should run on different type of computer system. A program is called *portable* if it can be transferred from one system to another with ease. This feature helps a lot in research work for easy movement of programs. High level language programs are more portable than the programs in assembly language.

(*v*) **Robustness.** A program is called *robust* if it provides meaningful results for all inputs (correct or incorrect). If correct data is supplied at run time, it will provide the correct result. In case the entered data is incorrect, the robust program gives an appropriate message with no run time errors.

(*vi*) **User friendly.** A program that can be easily understood even by a novice is called user friendly. This characteristic makes the program easy to modify if the need arises. Appropriate messages for input data and with the display of result make the program easily understandable.

(*vii*) **Self-documenting code.** The source code which uses suitable names for the identifiers is called self-documenting code. A cryptic (difficult to understand) name for an identifier makes the program complex and difficult to debug later on (even the programmer may forget the purpose of the identifier). So a good program must have self-documenting code.

2.3 MODULAR APPROACH

Breaking down of a problem into smaller independent pieces (modules) helps us to focus on a particular module of the problem more easily without worrying about the entire problem. No processing outside the module should affect the processing inside the module. It should have only one entry point and one exit point. We can easily modify a module without affecting the other modules. Using this approach the writing, debugging and testing of programs becomes easier than a monolithic program. A *modular* program is readable and easily modifiable. Once we have checked that all the modules are working properly, these are linked together by writing the main module. The main module activates the various modules in a predetermined order. For example, the following figure illustrates this concept :

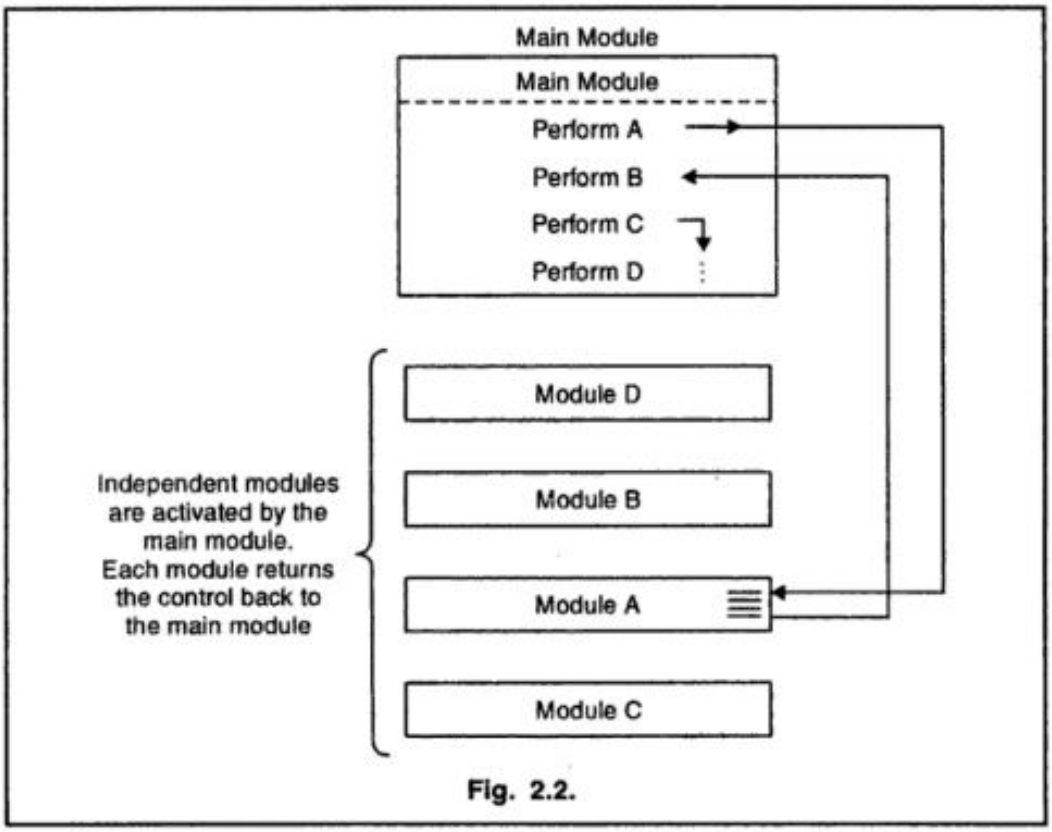

Fig. 2.2.

It must be noted that each module can be further broken into other submodules.

2.3.1 Characteristics of Modular Approach

(*i*) The problem to be solved is broken down into major components, each of which is again broken down if required. So the process involves working from the most general, down to the most specific.

(*ii*) There is one entry and one exit point for each module.

(*iii*) In general each module should not be more than half a page long. If not so, it should be split into two or more submodules.

(*iv*) Two-way decision statement are based on IF..THEN, IF..THEN..ELSE, and nested IF structures.

(*v*) The loops are based on the consistent use of WHILE..DO and REPEAT..UNTIL loop structures.

2.3.2 Advantages of Modular Approach

(*i*) Some modules can be used in many different problems.

(*ii*) Modules being small units can be easily tested and debugged.

(*iii*) Program maintenance is easy as the malfunctioning module can be quickly identified and corrected.

(*iv*) The large project can be easily finished by dividing the modules to different programmers.

(v) The complex modules can be handled by experienced programmers and the simple modules by junior ones.

(vi) Each module can be tested independently.

(vii) The unfinished work of a programmer (due to some unavoidable circumstances) can be easily taken over by someone else.

(viii) A large problem can be easily monitored and controlled.

(ix) This approach is more reliable.

(x) Modules are quite helpful in clarification of the interfaces between major parts of the problem.

2.4 PROGRAMMING STYLE

Programming style is concerned with writing good programs. It is upto a programmer to have his/her own style of writing a good program. There are different sets of guidelines that make programs readable, efficient, reliable, easy to use, flexible and portable. All these guidelines together form a good programming style. The guidelines for making a program readable are :

(i) Clarity and simplicity of expressions

(ii) Use of proper names for identifiers.

(iii) Inclusion of comments.

(iv) Inclusion of blank spaces and blank lines.

(v) Indentation and formatting.

2.5 CLARITY AND SIMPLICITY OF EXPRESSIONS

The high level languages allow a programmer to form arithmetic and logical expressions using arithmetic, logical and relational operators. The operator precedence rules help in elimination of any ambiguity. The operator precedence rules must be kept in mind while writing the expressions. If we do not follow the rules, the expression may be misinterpreted. Therefore, whenever we are in doubt, we must make use of parenthesis *i.e.* ( ) to enclose the subexpression. Note that two arithmetic operators should never be used in succession. They must be separated by an operand or parenthesis.

Use of parenthesis for enclosing subexpressions improves the readability and clarity of expressions. However, if there are many levels of parenthesis, then there may be difficulty in the understanding of the expression. In such cases we may split the expression into subexpressions using temporary variables, which are easy to understand.

The expressions used in the program should be simple so that their purpose can be easily understood. All the program statements should be written in readable form. All the high level languages have the standard built in functions also. So, the use of these standard functions also makes the expressions more readable.

The clarity and simplicity of expressions has been followed, throughout UNIT 3 : INTRODUCTION TO PROGRAMMING IN C++, by taking into account the following points :

(i) No programming tricks have been used.

(ii) Clarity has been preferred in expressions instead of little gain in machine run time.

(*iii*) Standard C++ functions have been used in expressions (wherever possible) to increase the readability.

2.6 USE OF PROPER NAMES FOR IDENTIFIERS

Identifier is used to assign a name to some program element *i.e.*, variable, constant etc. These are created by the programmer. Therefore, it is suggested that meaningful and informative names should be used for identifiers. This activity is known as **self documentation.** For example, the following variable names can be used (if the syntax of the language permits) :

avg_marks
total_salary
roll_number

The meaningful names for variables enhance the clarity and in turn make the statements easy to understand. Misleading variable names make the program cryptic, so it must be avoided. Also do not use similar looking names.

Similarly, meaningful identifier names can also be used for constant values. If the value of the constant is to be changed in later versions of the program, then only one change is enough, at the place of declaration.

The function names should also be selected in accordance with their expected objectives. For example functions that find average, maximum and minimum should be given names AVERAGE, MAX and MIN respectively. Thus the symbolic names used for identifiers increase program readability.

2.7 COMMENTS

Comments are one of the most important parts of a program. These are used for internal documentation of program that are ignored by the compiler. Comments are useful in understanding the program and help us in debugging (removing errors), maintaining and modifying the program. However, it is found that comments are often not used to save time of writing. Generally programmers think that these will be written later on (*i.e.*, at the end of programming) but the details of program are not so easy to remember at all times. Therefore, it is better to include comments while coding it, as at this time the program details are quite clear. There are two types of comments viz. prologue comments and explanatory comments.

Prologue Comments

These are written before the program, informing about the author, date, purpose of the program, what it is doing, how it is doing, how the program will be used, names of input and output files. Similarly, we may include such comments in the beginning of each function.

Explanatory Comments

These are written for explaining that part of the program which performs some complex computations, difficult to understand. It is a good idea to include comments before every control structure (*e.g.*, selection or iteration statements) explaining what that block is doing and what is the purpose of parameters/variables used in that block. A distinction between global variables and local variables should be clearly specified.

Excessive Comments

Inclusion of comments in a program is quite useful. However, excessive comments may make the program unnecessarily long and unreadable. Adding comments to code which is self-explanatory is not good programming feature. Comments should be clear, concise and must provide additional information which is not self-evident by the program code. The program code and the associated comments should agree with each other. Misleading comments make the program cryptic (difficult to understand). Always remember that modification in program code requires modification in associated comments also.

Note. In C++, comments are given by using // (double slash) or /*....*/

2.8 INDENTATION

Indentation is used for improving the readability of programs. The statements should be indented in such a way that nesting of groups of control statements is clearly visible. For example, the C++ statement given below :

```
if (a > b) big = a;
else
   big = b;
```

can be indented as follows :

```
if (a > b)
   big = a;
else
   big = b;
```

The general guidelines for *indentation* are given below :

(*i*) One instruction must be placed in one line as this makes subsequent modifications easier.

(*ii*) Each braces pair { and } must begin in the same column. It helps us in finding errors related to mismatched { and } pair. The following C++ statement illustrates this :

```
if (a > b)
   {
      if (a > c)
         big = a;
      else
         big = c;
   }
else
   {
      if (b > c)
         big = b;
      else
         big = c;
   }
```

(*iii*) All variables of one type should be declared in one line.

(*iv*) Each constant should be defined on a separate line.

(*v*) Statements enclosed within { } pair should be indented 5 places.

There are no precise rules of indentation. However, every programmer should evolve his/her own indentation habits and follow those so that all the programs structure (body) are clear and easily readable.

2.9 DOCUMENTATION AND PROGRAM MAINTENANCE

In the educational environment, students are asked to write programs only for learning purposes. The problems assigned to students are well defined and have already been analyzed. The program sizes are also small. Generally there are no users of the programs written by the students. A program once written and graded by the teacher is thrown away. Thus, the student environment is not the realistic one. This is not the case with commercial/industrial environment. The professional programmers write the programs and these are used by other persons who are not programmers. A maintenance team also looks after these programs. Therefore, it is very important that a good documentation should be there for users and maintenance programmers.

2.9.1 Documentation

Documentation of a program consists of written description of program's specification, its design, coding, operating procedures etc. It is quite useful to users and maintenance programmers but this activity is most disliked by programmers and therefore, generally, it is done in a very much haphazard way. As a result, documentation is either incomplete or inaccurate, making the users and maintenance programmers irritated and annoyed over it.

Documentation can be broadly classified into two types :

(1) *Documentation for users*

(2) *Documentation for maintenance programmers.*

(1) Documentation for Users

It is meant for the users of the program and describes how to use the program. If the user documentation is good, program can be used by users without any problem. If documentation is bad, then the program even performing some useful thing becomes unpopular among users. User documentation describes how to use a given program and hardware/software requirements for executing (running) it. It may consist of the following :

(*a*) Program description and its purpose.

(*b*) The method used for solving the problem as represented by the program. List of reference material, if any.

(*c*) Data input formats for executing the program.

(*d*) Commands required to start the operation of program and the type of interactions required from the user during program execution. The types of errors that may occur, response produced by the program and user response. All these are part of user interface.

(*e*) Detailed explanation of messages produced by the program during it's execution.

(*f*) Output formats produced by the program.

(*g*) Sample runs of program with sample input (all possible sets of data), user interactions required during program execution and its output.

(*h*) Description of program's performance, *e.g.*, response time, accuracy of computations etc.

(*i*) Limitations of the program *e.g.*, limitations on array size (both one and two dimensional arrays). When a problem involves a technique of mathematical/statistical modelling, limitations may be imposed by the underlying model.

The following hardware/software requirements must be specified in the user document :

(*a*) Type of computer for running the program (PC, minicomputer, super computer etc.).

(*b*) Operating system (*e.g.* MS-DOS, WINDOWS, UNIX, Linux etc.) and any other software such as database management system/graphics/simulation software needed for program execution.

(*c*) Hard disk and magnetic tapes with their sizes required.

(*d*) Input/Output devices required (*e.g.*, mouse, printer, plotter etc.).

(*e*) Size of main memory required for program execution.

(*f*) Any special hardware required such as mathematical coprocessor, network related device etc.

In addition to the above mentioned list of user documentation, the user requirements may vary depending on the program complexity, and its operating environment. The user documentation should not contain description about the internal working of the program.

(2) Documentation for Maintenance Programmers

It is also known as technical documentation and used for program modification at some later stages. It consists of :

(*a*) Internal documentation

(*b*) External documentation

(*a*) **Internal documentation.** It refers to the program listing itself, comments and *self-documenting code.* The term self-documentation code means that the program code should have meaningful variable names, *e.g.*, SUM, NUMBER, AVERAGE etc. This document is the most authentic and describes what it does and how it does that. A program is readable and understandable if it is well structured, properly formatted with blank spaces, blank lines and indentation, with meaningful variable names and comments etc. Thus, program listing is most important internal documentation and can be utilized by the maintenance programmers for modification.

(*b*) **External documentation.** As explained above, the internal documentation is enough for the purpose of program modification. However, when a program is very large, a separate and more detailed documentation outside the body of executable code is needed.

The external documentation may consist of the following :

(*i*) User documentation may optionally be included here for describing about the user needs to the programmer.

(*ii*) Detailed description of the problem, algorithm used for problem solving and program design.

(*iii*) Flowchart or Pseudocode of the complete solution.

(*iv*) Description of formats of each input data items, its purpose and type, format of output produced by the program and its purpose.

(*v*) List of important variables, their types, meaning and significance in the program.

(*vi*) List of all functions (or subprograms) called by the program. Brief description of each function, its flowchart or pseudocode. Description of arguments (parameters) and interaction with other modules.

(*vii*) List of library functions used within the program with the description of arguments.

(*viii*) Description of files used in the program and their purpose.

(*ix*) Description of numerical accuracy in case of mathematical/statistical problems.

(*x*) Language used for coding the program and compiler version used.

(*xi*) Description of any non-standard feature of the language inside the program.

(*xii*) Description about testing and program maintenance.

Production of Documentation

Documentation of a program must start with the start of problem definition. It is very difficult to think about the documentation at the end of program testing and debugging.

User documentation involves the functional specifications of the program. After the detailed analysis of the problem and user's requirements, functional specifications must be prepared. This document helps in algorithm/program design. In most of the cases, the program keeps changing until the final version of the program is produced. So, the user document also needs appropriate changes. However, if this document requires too many changes, it shows that the proper analysis of the problem has not been done.

Once the functional specifications are over, the programmer starts with the algorithm development phase. At this time, the technical details should be recorded and technical documentation starts. The functional specifications and the technical details written so far are used as a reference guide by the programmer. This helps in avoiding discrepancy between the program and the user requirements.

The technical documentation is prepared alongwith program development and its coding. Sometimes, this document may require few changes in the part already written because of changes in program design. Too many changes in documentation means the program design was not appropriate. After the coding of algorithm is over, testing is done. The test data and result of using test data must be specified in technical documentation. A working program must ensure what it does, in its documentation.

For a program to be useful, it must have a good user and technical documentation. A user manual of good quality is needed for a useful program. Good technical documentation is required for better understanding and modification of a program, which ensures a long life. Unnecessary explanations of simple things can make the program difficult to understand, on the other side, if too little is explained then it may not help in understanding of the program.

2.9.2 Program Maintenance

Program maintenance includes all those activities which help in correcting or preventing errors in software. These include correcting coding and design errors, updating documentation and test data, and upgrading user's support. In other words, program maintenance involves enhancement, *i.e.*, adding, modifying or developing the code to accomplish the needs.

There are two main reasons for program maintenance, as given below :

(*i*) *Existence of bugs*

(*ii*) *Program modification*

(*i*) Existence of Bugs

A program is generally used after it has been tested/debugged and it is found in a satisfactory working order. However, there may be many obscure errors existing in the program that may not get detected at the debugging/testing phase. Many errors may be detected, once it is released for public use, in its early stage. One of the most important activity of maintenance programmers is to fix these errors. This activity continues throughout the life of the program but not as much as in the initial stage of the program release.

(*ii*) Program Modification

User requirements from a particular program are not static. The program maintenance is required to meet the changing requirements of users by way of adding new features or improving performance. In fact, programs should be written in such a way that frequent changes are not required. However, coding of such programs may not be possible, as it is not always possible to predict the future requirements.

There may be external factors such as new laws, new ideas, new products, or new computer facilities. For example, change in income tax rules, modification in salaries annually, time to time concessions given to persons travelling in trains.

Problems Faced in Program Maintenance

The program maintenance becomes difficult, when a program is to be modified by the person who is not the author of the program, *i.e.*, in case the original programmer is not available. Therefore, the study and understanding the logic of the original program takes time for it's modification. This process may involve more time, money, and there are many chances of making errors. So, in such situations, it may be easier and economical to rewrite a new program.

Alterations in the programs are always done manually. So, there is no surity that the modified code will function appropriately. It depends mostly on the programmer's ability to judge and then make changes so that the program works fine. The program must be thoroughly tested after modification and then implemented.

2.10 RUNNING AND DEBUGGING PROGRAMS

Running a computer program written in any high-level language requires several steps, as given below :

(*i*) Develop the source code (program).

(*ii*) Select a suitable filename to store the program.

(*iii*) Create the program (type the program) and save it under the filename decided in step (*ii*). This file is known as *source code file.*

(*iv*) Compile the source code. The file having the translated code is known as *object code file.* If any error(s) occur, debug and compile the program again.

(*v*) Link the object code with other library code that are needed for execution. The resulting code is called the *executable code.* If any error found during linking, correct them and compile the program again.

(*vi*) Run the executable code and get the results, if there are no errors then stop.

(*vii*) Debug the program, if error(s) is/are found in the result.

(*viii*) Go to step (*iv*) and repeat the steps again.

These steps are illustrated with the help of flowchart in figure 2.3. The exact steps depend upon the program environment and the compiler in use. But, they will resemble with the above explained steps.

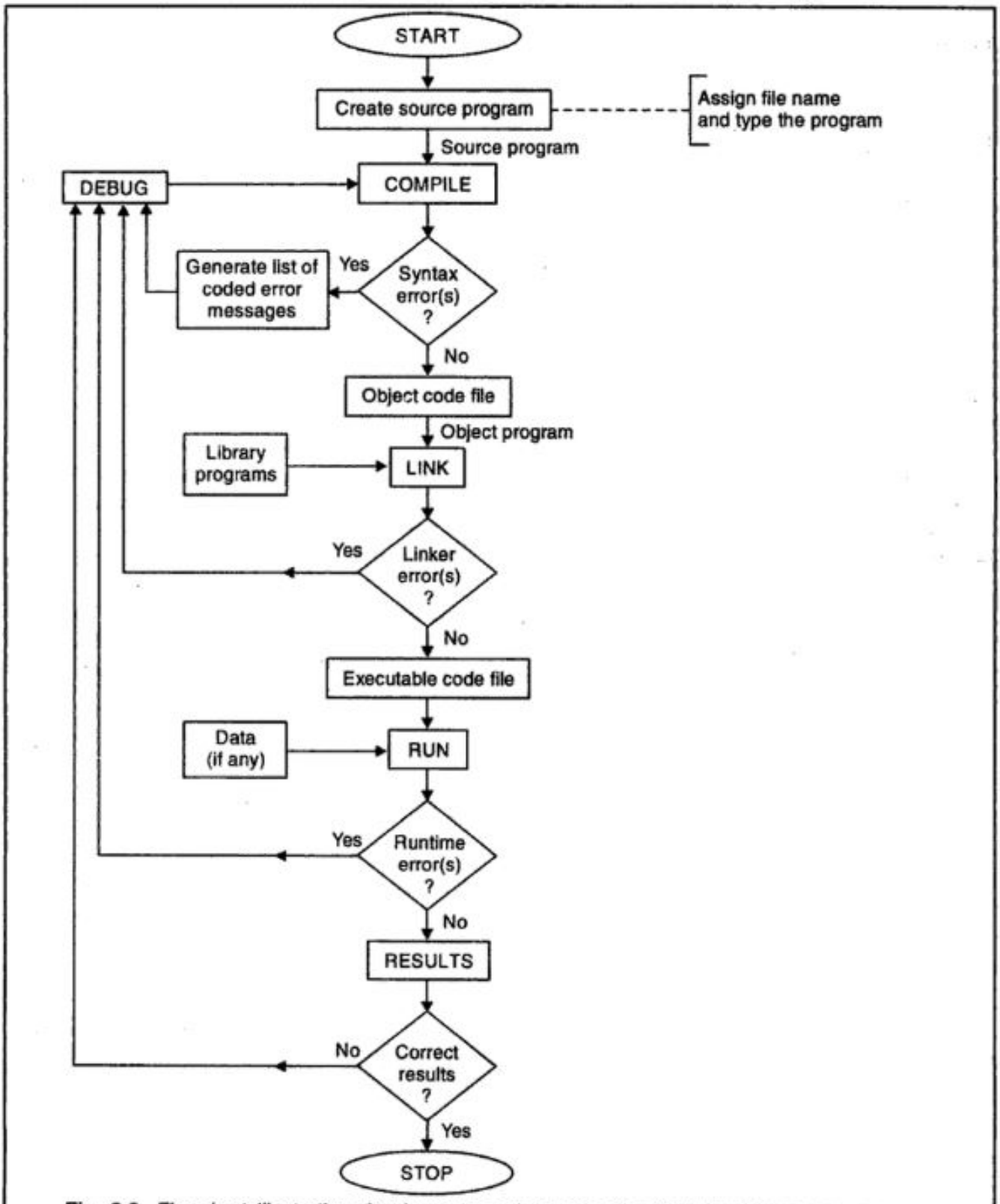

Fig. 2.3. *Flowchart illustrating development and running of a high level language program.*

It has been mentioned earlier that probability of making errors is reduced if a program is written using modular approach. Still, it is found, that errors may occur in the program. Until errors occur in the program, it is not usable. Therefore, all the errors in the program must be removed for running it.

Debugging

Debugging is the process of locating an error in the program and removing it. In the context of computers, an error is known as a **bug.** The word *bug* is used for any error which prevents normal functioning of the computer/program. Debugging a program is not an easy exercise. Generally, 50%–90% of total program development time is spent in debugging a program. Debugging a program depends on its environment in which the program is run. It depends on :

(*i*) the computer in use.
(*ii*) the language of the program.
(*iii*) the compiler used.
(*iv*) the operating system in use.
(*v*) debugging aids.
(*vi*) the problem faced.
(*vii*) the program being debugged.

A program may stop working due to several reasons. The reasons can be external or internal to the program. The external reasons could be hardware failure or operating system failure. The internal reasons are related to the program. These are either syntax errors or semantic errors (runtime errors or logical errors).

2.11 SYNTAX ERRORS

These errors occur when the grammatical rules of the programming language are not followed strictly in writing a program. These are similar to grammatical mistakes made in constructing sentences. Consider the following C++ statements :

```
sum : = a + b + c;
num  = 1000
x    = sqrt (num +);
```

All the above statements have syntax errors. The first statement has wrong assignment operator, in C++ the assignment operator is = instead of :=. Every C++ statement is separated by a ; (semicolon). It is missing after the second statement and, therefore, it is a syntax error. The third statement has missing operand after '+'. In general, syntax errors may be caused by misspelt words, missing or wrong punctuations, missing operands and operators, wrong construction of statements.

All the syntax errors are detected by the language compiler. The compiler produces a complete list of all syntax errors with error diagnostic messages. A single error may cause many error messages. For example, if we forget to declare a variable which is used in 10 statements, we get 10 error messages corresponding to these 10 statements. Removing this error *i.e.*, declaration of the variable will remove 10 error messages.

One should always try to write the syntax correctly the first time, as a syntax error wastes computing time and money, as well as programmer's time. With a little care it is preventable.

A program, after removing all syntax errors in it, is ready to be executed. However, there may be other errors in the program, which may prevent it from proper execution. Such errors are known as **semantic errors.** Semantic errors are either runtime errors or logical errors.

2.12 RUN-TIME ERRORS

If a program is syntactically correct, it is successfully compiled *i.e.*, translated into machine language. While executing the program it may behave abnormally due to presence of some errors other than syntax errors. The errors which appear during program execution are known as *run-time errors.* For example, *division by zero, square root of a negative number, integer/real arithmetic underflow or overflow, range errors, array indices out of bounds, non-existence of referenced file* (Input/Output error) etc. These can be avoided using the following suggestions :

(*i*) Echo the data *i.e.*, display the input data immediately.
(*ii*) Input validation—Range check.
(*iii*) Integer/real arithmetic check.
(*iv*) Never divide by zero.
(*v*) Check the parameter of **sqrt ( )** function.
(*vi*) Check for presence of files used in the program.

Note. The above suggestions have been followed in UNIT 3 : INTRODUCTION TO PROGRAMMING IN C++.

2.13 LOGICAL ERRORS

Once all syntax errors and runtime errors are removed, the program can execute without abnormal termination and can provide result. Therefore, with some input data (if required), a program is executed. However, the output produced may not be the required or there may not be any output. It means that errors exist in the program. All those errors which are not syntax errors or runtime errors are known as **logical errors.** A *logical error* is an error which occurs in a program by incorrect translation of the logic of the problem. These are programmer's mistakes and are most difficult to debug.

Logical errors may occur due to the following reasons :

(*i*) Wrong placement of statements.
(*ii*) Not initializing the variables.
(*iii*) Omission of parenthesis.
(*iv*) Occurrence of an infinite loop.
(*v*) Execution of loop a wrong number of times.
(*vi*) Errors associated with boundary value data.
(*vii*) Translation of correct algorithm into incorrect program.
(*viii*) Error in problem definition.

2.14 PROBLEM SOLVING METHODOLOGY AND TECHNIQUES

Computer problem-solving can be summed up in one word—*it is demanding !* It is a combination of many small parts put together in a complex way, and therefore difficult to understand. It requires much thought, careful planning, logical accuracy, continuous efforts, and attention to detail. Simultaneously it can be a challenging, exciting, and satisfying experience with a lot of room for personal creativity and expression. If computer problem-solving is approached in this spirit then the chances of success are very bright.

For solving a problem on a computer a set of explicit and unambiguous instructions is written in a programming language. This set of instructions is called a *program.* An algorithm (step by step procedure to solve a problem in unambiguous finite number of steps) written in a programming language is a program. So, an algorithm corresponds to a solution to a problem which is *independent* of any programming language.

Problem solving is a creative process which largely defies systematization and mechanization. Everyone acquires some problem-solving skills during his/her student life which he/she may or may not be aware of.

Some steps for problem solving improve the performance of the problem solver. No universal methods are available for it. Different people use different strategies. In simple words we can say logically that computer problem solving is about understanding.

2.15 UNDERSTANDING OF THE PROBLEM

When lot of efforts are made in understanding the problem we are dealing with, chances of success are also bright. We cannot hope to make useful progress in solving a problem until it is clear, what it is we are trying to solve. The preliminary investigation may be thought of as the *problem definition phase.* The problem definition defines what the problem is without any reference to the possible solutions. It is a simple statement, may be one to two pages and should sound like a problem. The problem definition should be in user language and it should be described from the user's point of view. It usually should not be defined in technical computer terms. As the analyst assigns the programs to different programmers module-wise, the programmers understand the problem given to them. The programmers define the problem of each program on a document and proceed for the next step. In simple words, a lot of care should be taken in working out precisely what must be dcne.

The problem solver should obtain information on the following three aspects of the problem after the analyses :

1. Input specification
2. Output specification
3. Special processing, if any.

1. Input Specifications

The input specifications should give the following information :

(*i*) Specific data values to be used as input in the program.

(*ii*) Input data format *i.e.*, order, spacing, accuracy and units.

(*iii*) The valid range of input data.

(*iv*) Restrictions, if any, on use of these data values and what to do if an input data is not accepted by the computer, should it be ignored or modified.

(*v*) The indication of end of input data (if specified by a special symbol).

2. Output Specifications

The output is obtained on executing a program. The output specifications must clearly define the values required and their formats etc. The output specifications must include the following information :

(*i*) The output data values required.

(*ii*) Output data format *i.e.*, precision (number of significant digits), accuracy, units, the position on the output sheet and suitable headings for making the output readable.

(*iii*) Amount of output required because the program has to be coded according to the number of output data values required.

3. Special Processing, if any

It means processing of input data under some conditions. If conditions are violated, certainly results are going to be incorrect. The processing under special condition(s) and the recovery action should be handled carefully. If the special processing conditions are ignored and left in the problem definition phase, it may be a costly affair later on.

So, in the problem definition phase, detailed information about input, output and special processing is gathered. These conditions are taken into consideration while solving the problem. The method of solution is not specified in this phase.

2.16 IDENTIFYING MINIMUM NUMBER OF INPUTS REQUIRED FOR OUTPUT

The term input means identifying initial data for the problem or program. This data must be present before any operation can be performed on it. Many a times, no data is required for a program because initial data may be generated within the program. So, a problem may have no or many inputs.

The two ways to supply initial values to variables are by using internal and external data.

(*i*) **Internal data** means that values are generated within the problem.

(*ii*) **External data** means that the required values of each element are to be inputted by the problem solver.

For example, consider the following C++ code segment for printing first 10 natural numbers :

```
for(num=1;  num<=10;  num++)
    cout<<num<<endl;
```

No input is required for printing the first 10 natural numbers, that is the desired output. The values for **num** are generated within the problem.

The following C++ code segment finds the largest of **n** elements and their sum using minimum number of input/variables :

```
cin>>n;        //input number of elements
sum=0;
cin>>num;      //input the first element
maximum=num;
     sum+=num;
for(i=1;  i<n;  i++)
     {
          cin>>num;
          if(num>maximum)
               maximum=num;
          sum+=num;
     }
cout<<"Largest number  =  "<<maximum<<"  sum="  <<sum<<endl;
```

Here, only one variable **num** is used for accepting values from the input device *i.e.*, keyboard for the elements and **n** for number of elements.

Therefore, the problem solver should understand the input data required for the program, and the output produced by the program. Many times the input data may be returned by some program module *i.e.*, a function or any other program called by the program we are using or some built in function available in the library of the language used for coding the problem.

So, the problem must be understood thoroughly for the desired output which helps a lot in identification of minimum number of inputs.

2.17 STEP BY STEP SOLUTION FOR THE PROBLEM

There are many ways to solve most of the problems and also many solutions to most of the problems. This situation makes the job of problem-solving a difficult task. When we have many ways to solve a problem it is usually difficult to recognize quickly which paths are likely to be fruitless and which paths may be productive.

A block often occurs after the problem definition phase, because people become concerned with details of the implementation *before* they have completely understood or worked out an implementation-independent solution. The problem solver should not be too concerned about detail. That can be taken into account when the complexity of the problem as a whole has been brought under control. The old computer proverb states, **"the sooner you start coding your program the longer it is going to take".**

An approach that often allows us to make a start on a problem is to take a specific example of the general problem we wish to solve and try to work out the mechanism that will allow us to solve this particular problem (*e.g.* if you want to find the top scorer in an examination, choose a particular set of marks and work out the mechanism for finding the highest marks in this set).

This approach of focusing on a particular problem can often give us a platform we need for making a start on the solution to the general problem. It is not always possible that the solution to a specific problem or a specific class of problems is also a solution to the general problem. We should specify our problem very carefully and try to establish whether or not the proposed algorithm (step by step procedure in a finite number of steps to solve a

problem) can meet those requirements. If there are any similarities between the current problem and other problems that we have solved or we have seen solved, we should be aware of it. In trying to get a better solution to a problem, sometimes too much study of the existing solution or a similar problem forces us down the same reasoning path (which may not be the best) and to the same dead end. Therefore, a better and wiser way to get a better solution to a problem is, try to solve the problem *independently*.

Any problem we want to solve should be viewed from a variety of angles. When all aspects of the problem have been seen, one should start solving it. Sometimes, in some cases it is assumed that we have already solved the problem and then try to work backwards to the starting conditions. The most crucial thing of all in developing problem-solving skills is practice.

Probably the most widely known and most often used principle for problem-solving is the *divide-and-conquer* strategy. The given problem is divided into two or more subproblems which can hopefully be solved more efficiently by the same technique. If it is possible to continue in this way we will finally reach the stage where the subproblems are small enough to be solved without further splitting.

This way of breaking down the solution to a problem has been widely used with searching, selection and sorting algorithms.

2.18 BREAKING DOWN SOLUTION INTO SIMPLE STEPS

The primary goal in computer problem-solving is an algorithm which helps in implementation of a correct and efficient computer program. Once we have defined the problem to be solved and have an idea of how to solve it, we can use the powerful techniques for designing algorithms. For successful design of an algorithm we must have proper understanding of the inherent complexity of the problem that require computer solution. A problem solver can properly focus on a very limited part of the logic or instructions. A technique which is very useful for algorithm design taking into account the limited part of it is known as **top-down design** or **stepwise refinement.**

The top-down design approach helps in bringing a vague outline solution to a precisely defined algorithm and program implementation. It provides us a way of handling the inherent logical complexity and detail, most commonly found in computer algorithms.

For solving any problem first of all we must have at least the broadest of outlines of a solution. Sometimes this might demand a lot more investigation into the problem while at other times the problem description may in itself give the necessary starting point for top-down design. The general outline may consist of a single statement or a set of statements.

Top-down design suggests that we take the general statements that we have about the solution, one at a time, and break them down into a set of more exactly defined subtasks. These subtasks should more accurately describe about reaching the final goal. When we split a task into subtasks, we must exactly define the way in which the subtasks will interact with each other. By doing so the overall structure of the solution to the problem can be preserved. The preservation of the overall structure in the solution to a problem makes the algorithm comprehensible and helps in proving the correctness of the solution.

The process of repeatedly breaking a task down into subtasks and then each subtask into still smaller subtasks must continue until we reach at the subtasks that can be coded as program statements. In most of the cases we need to go down to two or three levels but for large software projects this is not true. Figure 2.4 shows the schematic breakdown of a problem :

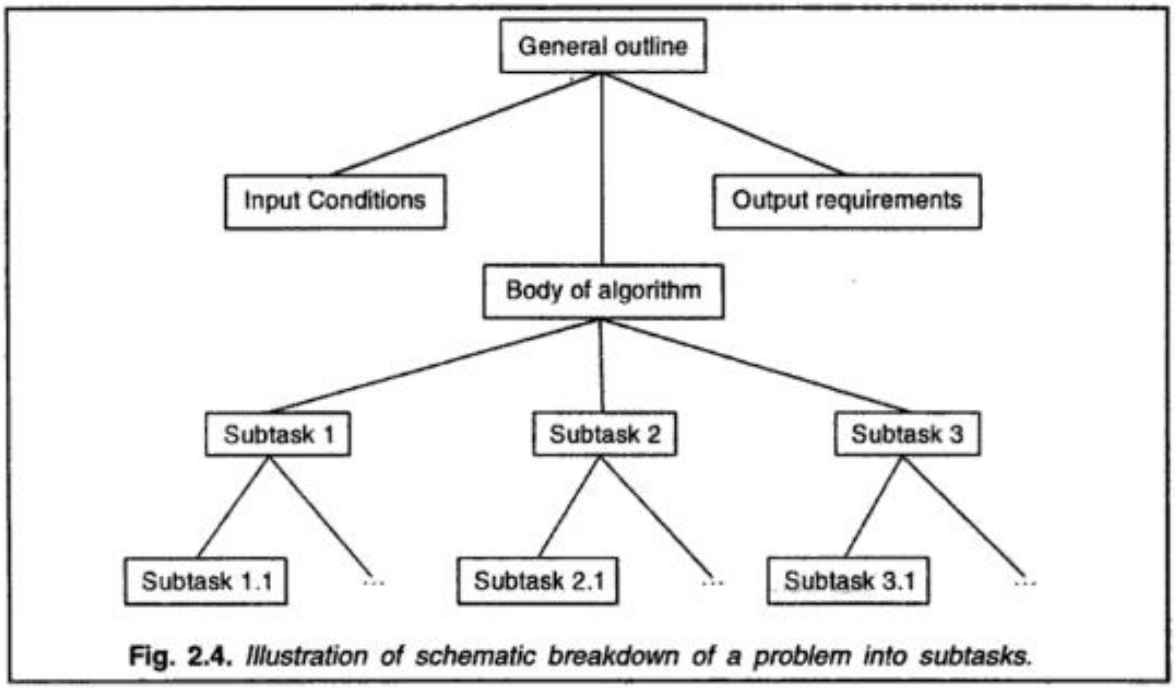

Fig. 2.4. *Illustration of schematic breakdown of a problem into subtasks.*

The process of breaking down the solution to a problem into subtasks in the manner described gives an implementable set of subtasks which can be easily coded into languages like C, C++, VB etc. There can therefore be a smooth and natural interface between the stepwise refined algorithm and the actual program code—a highly desirable situation for keeping the implementation task very simple.

The following simple example illustrates topdown design with modules, for the problem of finding combinations of N objects taken R at a time. For solving the problem, the top level statement could be :

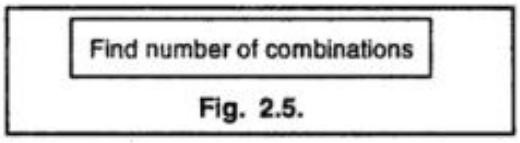

Fig. 2.5.

This statement can be further broken into the following four subproblem statements :

1. Print title of what the program is going to do before asking the user to type numbers. Call this module as **Printtitle.**
2. Get integers N and R such that R <= N. Call this module as **Readnumbers.**
3. Calculate the number of combinations of N objects taken R at a time which is $\frac{N!}{R!(N-R)!}$. Call this module as **Noofcombinations.**

4. Print the calculated number of combinations with appropriate heading. Call this module as **Displayresult.**

With this refinement of top level, we have the following hierarchical structure :

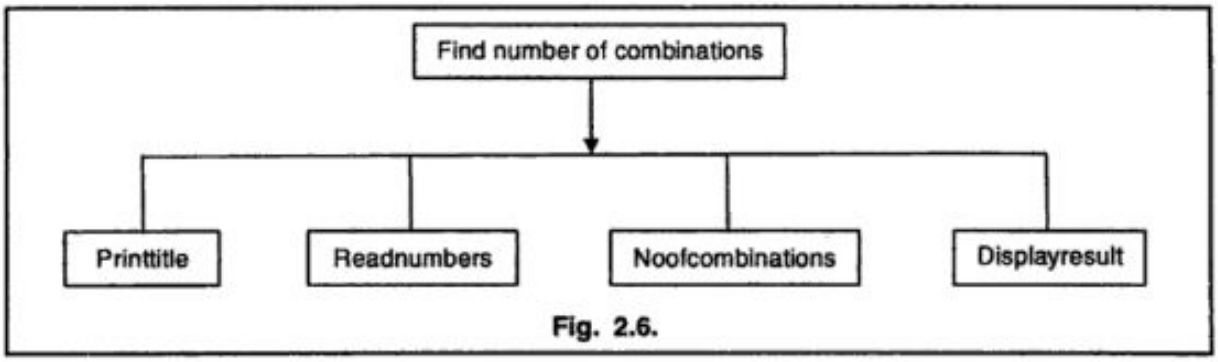

Fig. 2.6.

The above shown modules can be further refined as follows :

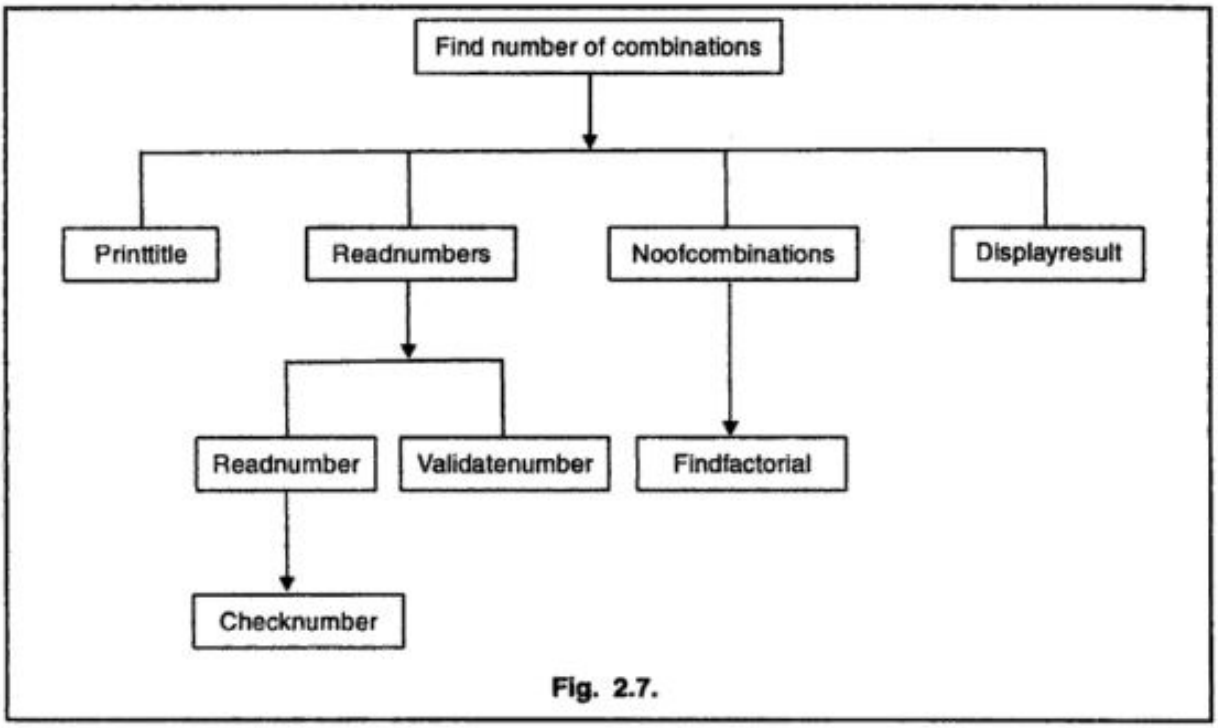

Fig. 2.7.

2.19 IDENTIFICATION OF ARITHMETIC AND LOGICAL OPERATIONS REQUIRED FOR SOLUTION

Most of the computational problems involve arithmetic and logical operations. The efficiency of a program is measured by its execution time and memory space. Generally a program which takes less execution time, takes more memory space and a program that takes less memory space is slower. So, a compromise can be made between the two factors.

It is not necessary that every problem includes both the arithmetic and logical operations for the solution. Some problems include only arithmetic operations. For example, the following C++ code finds the average of three numbers :

```
float   num1,num2,num3,avg;
cout<<"Enter the three numbers\n";
cin>>num1>>num2>>num3;
avg=(num1+num2+num3)/3.0;
cout<<"Average="<<avg;
```

Here, only the arithmetic operations (addition and division) are needed for the solution.

Some problems include only logical operations. For example, the following C++ code checks a year for leap year :

```
int year;
cout<<"Enter the year\n";
cin>>year;
if(((year%4 == 0) && (year%100 != 0)) || (year%400==0))
    cout<<year<<" is a leap year\n";
else
    cout<<year<<" is not a leap year\n";
```

Here, the logical operators !, && and || are used for the logical operations.

Some problems include both arithmetic and logical operations for their solution. For example, the following C++ code finds the type and area of a triangle.

```
cin>>a>>b>>c;
if(((a+b)>c) && ((b+c)>a) && ((c+a)>b))
    {
        if((a==b) && (b==c))
            cout<<"Equilateral triangle\n";
        else
            {
                if((a==b)||(b==c)||(c==a))
                    cout<<"Isosceles  triangle\n";
                else
                    cout<<"Scalene  triangle\n";
            }
        s=(a+b+c)/2.0;
        area=sqrt(s*(s-a)*(s-b)*(s-c));
        cout<<"\nArea = "<<area<<" sq.units\n";
    }
else
    cout<<"Triangle is not possible\n";
```

So, from above examples it is clear that the identification of arithmetic and logical operations required for solution of problems plays an important role in problem solving. It depends on the problem solver's efficiency to identify these appropriately.

2.20 USING CONTROL STRUCTURE

Any problem can be solved by using the three control structures given below :

(*i*) Sequence control structure

(*ii*) Conditional control

(*iii*) Looping

It depends on the nature of the problem whether to use all of the above mentioned control structures or some of these.

2.20.1 Sequence Control Structure

The sequence control structure selects all the steps, one followed by another *e.g.*, if there are n statements or steps say statement1, statement2, ----, statementn, then these are executed in the sequence —— statement1 followed by statement2 and so on till the last statement is executed. Following figure shows sequence control structure :

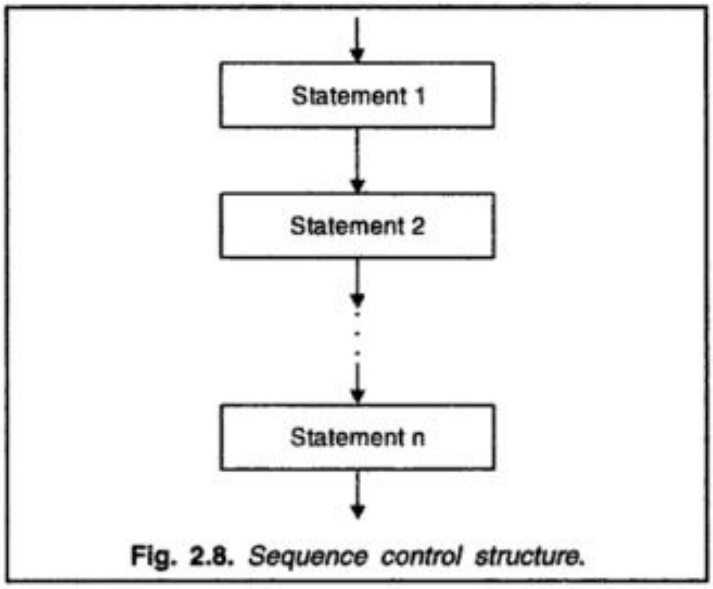

Fig. 2.8. *Sequence control structure.*

2.20.2 Conditional Control

The conditional control structure selects one or more steps for execution depending upon a given condition being true or false. For example, consider the following binary decision structure (IF–THEN–ELSE) :

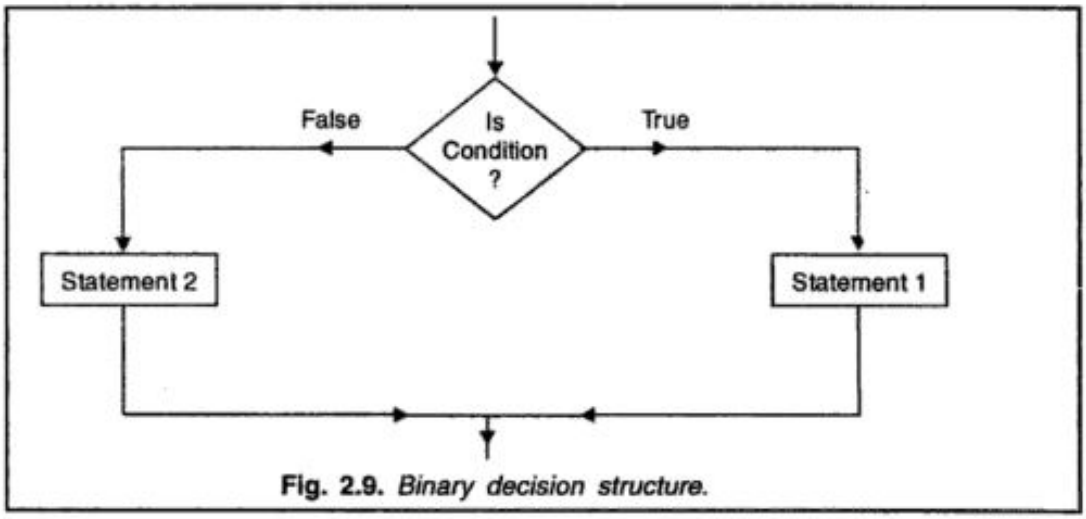

Fig. 2.9. *Binary decision structure.*

This structure first evaluates a logical expression (an expression containing the relational symbols such as <, <=, =, ≠, >, >=. This logical expression is also known as a condition. If this condition happens to be true then Statement 1 is to be executed and if this condition is false then Statement 2 is to be executed.

For example, in C++ language **if-else**, Nested **if**, **switch..case..default**, Nested **switch..case** statements can be used to implement conditional control.

Case is an important version of selection (more than a single yes–or–no decision). The following figure illustrates **variation on selection : the case control structure.**

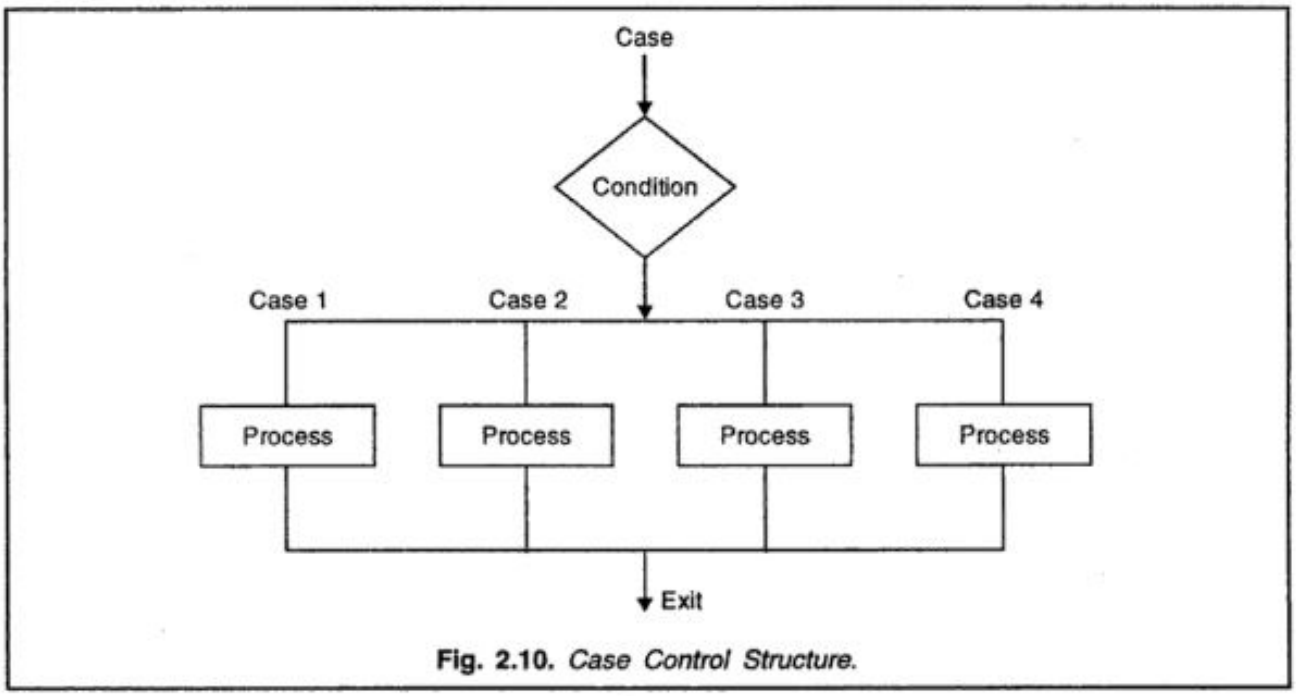

Fig. 2.10. *Case Control Structure.*

2.20.3 Looping (finite and infinite)

The looping control structure executes a statement a number of times depending on the value of the boolean expression. Following figures show two types of looping structures :

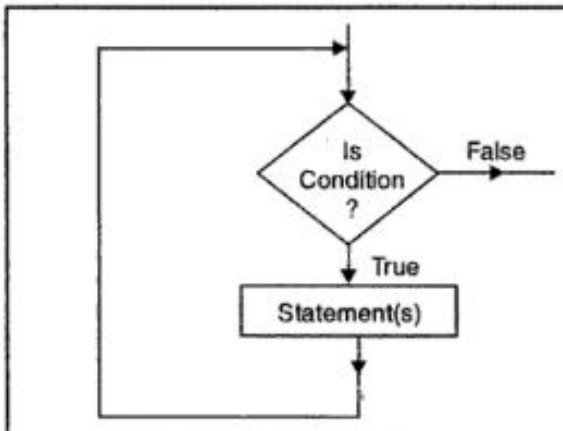

Fig. 2.11. *Looping Structure.*

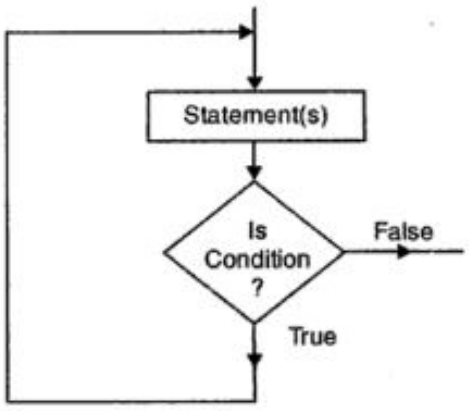

Fig. 2.12. *Looping Structure.*

In C++ language, **while, do..while, for** and Nested loops can be used to implement looping. Looping is also known as Repetitions or Iterations.

After refinement of a subtask we can get something that can be realized as an iterative construct. It can be easily implemented if we are aware of the basic structure of all loops. To construct any loop we must take into account three things :

(*i*) the initial conditions that need to apply *before* the loop starts to execute.

(*ii*) *the invariant relation* that must apply after each iteration of the loop.

(*iii*) the conditions under which the iterative process must *terminate.*

For example, we can find the sum of first **n** natural numbers in C++ using the three different looping structures (**n** is inputted by the user) :

while loop used for finding the sum of first *n* natural numbers :

```
i = 1;
sum = 0;
while(i <= n)
{
   sum = sum + i;
   i = i + 1;
}
```

This loop terminates after *n* iterations.

do..while loop used for finding the sum of first *n* natural numbers :

```
i = 1;
sum = 0;
do
{
   sum = sum + i;
   i = i + 1;
}while (i <= n);
```

This loop terminates after *n* iterations.

for loop used for finding the sum of first *n* natural numbers :

```
sum = 0;
for(i = 1; i<= n; i++)
    sum = sum + i;
```

This loop terminates unconditionally after *n* iterations.

Consider the following example,

```
while(num >0 && num < 50)
    {
        ....
        ....
        ....
    }
```

With loops of this type it cannot be directly determined in advance how many iterations there will be before the loop will terminate. In fact there is no guarantee that loops of this type will terminate at all. In these cases the responsibility for making sure that the loop will terminate rests with the algorithm designer.

An infinite looping in C++ can be constructed as follows :

```
flag = 1;
while(flag)
```

```
{
    ....
    ....
    ....
}
```

Every problem must terminate in a finite number of steps, so it is the duty of the algorithm designer to set up the termination of a loop (if exists inside a problem). For example, by forcing the condition under which the loop will continue to iterate to become false.

2.21 SOFTWARE DEVELOPMENT STEPS

Every type of software (pre-written software, or a customized software, or a public domain software) has to be developed by someone before we can use it. Software or program must be understood properly before its development and use. **A program is a list of instructions that the computer must follow in order to process data into information.** The instructions consist of statements used in a programming language, such as C or C++. Examples are programs that do word processing, desktop publishing, or railway reservation.

The decision whether to buy or develop a program forms part of Phase 4 in the systems development life cycle. Figure 2.13 illustrates this. Once the decision is made to develop a new system, the programmer starts his/her work.

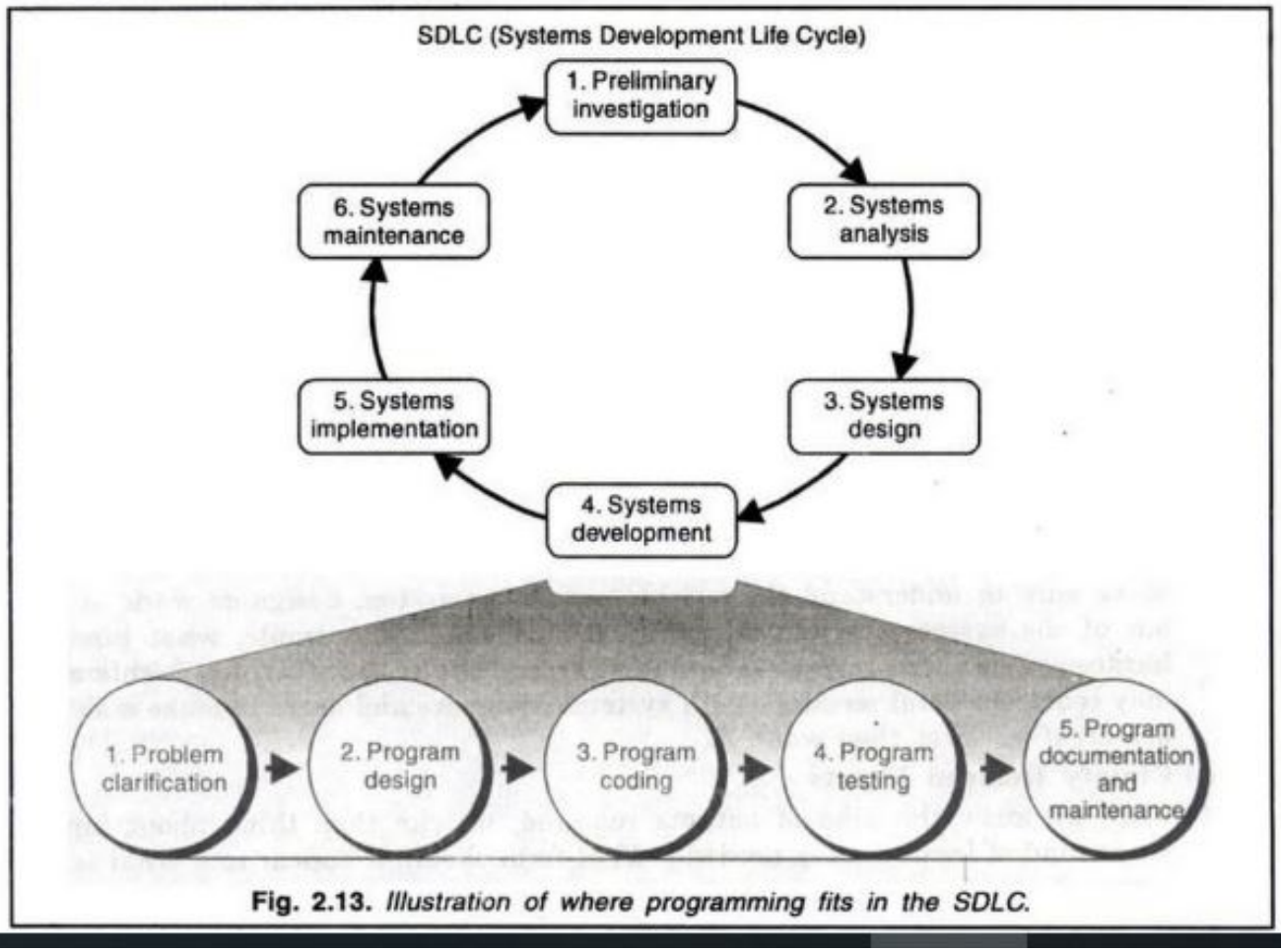

Fig. 2.13. *Illustration of where programming fits in the SDLC.*

The Phase 4 of the six-phase SDLC includes a five-step procedure of its own as shown in the bottom of figure 2.13. These five steps constitute the problem-solving or software development process known as programming. **Programming also known as software engineering, is a multistep process for creating that list of instructions** (*i.e.*, a program for the computer).

The five steps are given below :

1. Clarify the problem—include needed output, input, processing requirements.
2. Design a solution—use modeling tools to chart the program.
3. Code the program—use a programming language's syntax, or rules, to write the program.
4. Test the program—get rid of any logic errors, or "bugs", in the program ("debug" it).
5. Document and maintain the program—include written instructions for users, explanation of the program, and operating instructions.

Coding—sitting at the keyboard and typing words into a computer—is what many people imagine programming to be. As we see, however, it is only one of the five steps. Coding consists of translating the logic requirements into a programming language—the letters, numbers and symbols that make up the program.

1. Clarify the Problem

The *problem clarification* step consists of six sub-steps—clarifying program objectives and users, outputs, inputs and processing tasks; studying the feasibility of the program; and documenting the analysis. Let us consider these six sub-steps.

(*i*) Clarify Objectives and Users

We solve problems all the time. A problem might be deciding whether to take a required science course this term or next, or selecting classes that allow us also to fit a job into our schedule. In such cases, we are specifying our *objectives*. Programming works the same way. We need to write a statement of the objectives we are trying to accomplish—the problem we are trying to solve. If the problem is that our company's systems analysts have designed a new computer-based payroll processing program and brought it to us as the programmer, we need to clarify the programming needs.

We also need to make sure us know who the users of the program will be. Will they be people inside the company, outside, or both ? What kind of skills will they bring ?

(*ii*) Clarify Desired Outputs

Make sure to understand the outputs—what the system designers want to get out of the system—before we specify the inputs. For example, what kind of hardcopy is wanted ? What information should the outputs include ? This step may require several meetings with systems designers and users to make sure we are creating what they want.

(*iii*) Clarify Desired Inputs

Once we know the kind of outputs required, we can then think about input. What kind of input data is needed ? What form should it appear in ? What is its source ?

(*iv*) **Clarify the Desired Processing**

Here we make sure to understand the processing tasks that must occur in order for input data to be processed into output data.

(*v*) **Double-Check the Feasibility of Implementing the Program**

Is the kind of program we are supposed to create feasible within the present budget ? Will it require hiring a lot more staff ? Will it take too long to accomplish ?

Sometimes programmers decide they can buy an existing program and modify it rather than write it from scratch.

(*vi*) **Document the Analysis**

Throughout program clarification, programmers must document everything they do. This includes writing objective specifications of the entire process being described.

2. Design the Program

Assuming the decision is to make, or custom-write, the program, we then move on to design the solution specified by the systems analysts. In the ***program design*** **step, the software is designed in three mini-steps. First, the program logic is determined through a top-down approach and modularization, using a hierarchy chart. Then it is designed in detail, either in narrative form, using pseudocode, or graphically, using flowcharts.**

Today most programmers use a design approach called structured programming. ***Structured programming*** **takes a top-down approach that breaks programs into modular forms. It also uses standard logic tools called control structures (sequential, selection, case and iteration).**

More about it in Chapter-6.

3. Code the Program

Once the design has been developed, the actual writing of the program begins. **Writing the program is called *coding*.** Coding is what many people think of when they think of programming, although it is only one of the five steps. Coding consists of translating the logic requirements from pseudocode or flowcharts into a programming language—the letters, numbers, and symbols that make up the program.

(*i*) **Select the Appropriate Programming Language**

A ***programming language*** **is a set of rules that tells the computer what operations to do.** Examples of well-known programming languages are C, C++, COBOL, Visual Basic and JAVA. These are called "high-level languages".

Not all languages are appropriate for all uses. Some, for example, have strengths in mathematical and statistical processing. Others are more appropriate for database management. Thus, in choosing the language, we need to consider what purpose the program is designed to serve and what languages are already being used in our organization or in our field.

(*ii*) **Follow the Syntax**

In order for a program to work, we have to follow the ***syntax*, the rules of the programming language.** Programming languages have their own grammar just as human languages do. But computers are probably a lot less forgiving if we use these rules incorrectly.

4. Test the Program

***Program testing* involves running various tests and then running real-world data to make sure the program works.** Two principal activities are *desk-checking* and *debugging*. These steps are known as *alpha-testing*.

(*i*) **Perform Desk-Checking**

***Desk-checking* is simply reading through, or checking, the program to make sure that it's free of errors and that the logic works.** In other words, desk-checking is like proofreading. This step could be taken before the program is actually run on a computer.

(*ii*) **Debug the Program**

Once the program has been desk-checked, further errors, or "bugs", will doubtless surface. To ***debug* means to detect, locate, and remove all errors in a computer program.** Mistakes may be syntax errors or logical errors. ***Syntax errors* are caused by typographical errors and incorrect use of the programming language. *Logic errors* are caused by incorrect use of control structures.** Programs called *diagnostics* exist to check program syntax and display syntax-error messages. Diagnostic programs thus help identify and solve problems.

(*iii*) **Run Real-World Data**

After desk-checking and debugging, the program may run fine—in the laboratory. However, it needs to be tested with real data; this is called *beta testing*. Indeed, it is even advisable to test the program with *bad data*—data that is faulty, incomplete, or in overwhelming quantities—to see if you can make the system crash. Many users, after all, may be far more heavy-handed, ignorant and careless than programmers have anticipated.

Several trials using different test data may be required before the programming team is satisfied that the program can be released. Even then, some bugs may persist, because there comes a point where the pursuit of errors is uneconomical. This is one reason why many users are nervous about using the first version (version 1.0) of a commercial software package.

5. Document and Maintain the Program

Writing the program documentation is the fifth step in programming. The resulting ***documentation* consists of written descriptions of what a program is and how to use it.** Documentation is not just an end-stage process of programming. It has been (or should have been) going on throughout all programming steps. Documentation is needed for people who will be using or be involved with the program in the future.

Documentation should be prepared for several different kinds of readers—users, operators and programmers.

(*i*) **Prepare User Documentation**

When we buy a commercial software package, such as a spreadsheet, we normally get a manual with it. This is *user documentation.*

(*ii*) **Prepare Operator Documentation**

The people who run large computers are called *computer operators.* Because they are not always programmers, they need to be told what to do when the program malfunctions. The *operator documentation* gives them this information.

(*iii*) **Write Programmer Documentation**

Long after the original programming team has disbanded, the program may still be in use. If, as is often the case, a fourth of the programming staff leaves every year, after 4 years there could be a whole new bunch of programmers who know nothing about the software. *Program documentation* helps train these newcomers and enables them to maintain the existing system.

(*iv*) **Maintain the Program**

Maintenance includes any activity designed to keep programs in working condition, error-free, and up to date—adjustments, replacements, repairs, measurements, tests and so on. The rapid changes in modern organizations—in products, marketing strategies, accounting systems, and so on—are bound to be reflected in their computer systems. Thus, maintenance is an important matter, and documentation must be available to help programmers make adjustments in existing systems.

REVIEW QUESTIONS AND EXERCISES

1. What is modular approach ? Explain.
2. What do you understand by clarity and simplicity of expressions ? Explain with the help of examples.
3. What is the purpose of using proper names for identifiers ? Explain.
4. Why do we use comments ?
5. What is the use of indentation ? Explain.
6. Differentiate between commenting and indenting.
7. Write an explanatory note on documentation.
8. Differentiate between internal and external documentation.
9. What is program maintenance ? Explain.
10. Write a short note on running and debugging of programs.
11. What are different types of errors ? Explain.
12. Differentiate between run time errors and logical errors.
13. What is meant by syntax errors ? Explain.
14. Explain in detail the various characteristics of a good program.

15. Understanding of the problem is most important. Explain it.
16. What is step-wise refinement ? Explain by giving a suitable example.
17. Explain the use of control structure in problem solving with suitable examples.
18. Write a short note on identifying minimum number of inputs required for output in context of problem solving.
19. Write a short note on identification of arithmetic and logical operations required for solution.
20. Write a short note on software development steps.

❑❑❑