CHAPTER 1

BASICS OF PROGRAMMING

1.1 Introduction

A computer is capable of performing calculations or manipulating data with exact step-by-step instructions. A computer program is a series of instructions to carry out a particular task written in a language that a computer can understand. The process of preparing and feeding the instructions into the computer for execution is referred as ***programming***. People, who use the computer for writing programs to solve various types of problems using computer are referred to as ***programmers***. Some of the popular programming languages are C, C++, Oracle, Python, etc.

Programming requires the skill of writing code that gives proper solution for a problem and others can read and easy to alter. Programming requires concept of creating skills. Programming should describe methods of design and construction, and the methods selected should be such that there should be a gradual development. The following are features of a good program.

- Run correctly and efficiently.
- Have a user-friendly interface.
- Be easy to read and understand.
- Be easy to debug .
- Be easy to modify.
- Be easy to maintain.

In a business environment, programming is a team effort. A programmer must always keep the project objectives in mind. The programs written by one team member must be complementary to those written by other team members. For example, a team member might write a program to collect the data and some other team member would then write a program to analyse the data and print a report.The objective of this chapter is to make familiar with general concepts in programming and its methodologies.

1.2 Problem Solving Phase

A problem/project needs programs to create, append and update the database, print data, permit on-line enquiry and so on. A programmer should identify all requirements to solve the problem.

Each problem should have the following descriptions.

- Type of programming language.
- Narration of the program describing the tasks to be performed.
- Frequency of processing (i.e., hourly, daily, weekly, etc.).

- Output and input of the program .
- Limitations and restrictions for the program.
- Detailed specifications.

Fig. 1.1 shows the steps adopted to develop a program in any programming language.

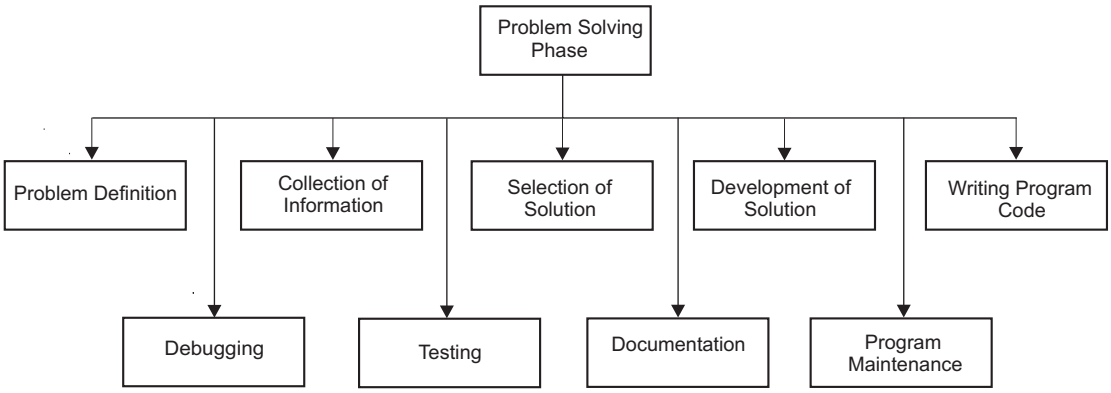


Fig. 1.1: *Steps adopted to develop a program.*

Problem Definition

Problem definition stage is an important step in writing a program. The problem should be well understood and should be defined in clear terms. This stage might involve a detailed discussion with the person for whom the program is made to solve the problem and get all the details regarding the problem. The various parameters and constraints of the problem are identified and the appropriate method/approach is selected for solving the problem.

Collection of Information

The programmer should select a best programming language to solve the problem and analyse the various limitations and restrictions necessary for the problem. Once the parameters, constraints and all useful information are collected, split the data into input and output data and the overall solution is laid out. The output data is the expected outcome of the program.

- Validate output from the program
  - o Tell the user about the generated output
  - o Format the data while printing the output
  - o Give the user meaningful feedback (return value) from functions.
- Validate user input to the program
  - o Tell the user what are expected to read as input
  - o Validate data while reading the input
  - o Avoid passing invalid data type from one function to another.

Although different programs have different components, a good place to start with most programs is to analyse the output, input and how they are processing and then identify important considerations in the design of the program logic.

Selection of Solution

Once the parameters, constraints, and all other useful information are identified, the best method/ approach is selected for solving the problem. If there is a unanimous choice for a problem, that choice can be applied for the solution to the problem. When the solution for a problem are abundant, the choice between them is made using criteria such as computer time required, space occupied in memory, accountability to error, etc. For some type of problems, if no satisfactory solution is known, a solution has to be developed, or the problem has to be simplified to the point where a solution becomes feasible.

Development of Algorithm

An ***algorithm*** is a well-defined sequence of instructions designed to solve a specific problem or perform a task. In order to develop an algorithm, identify the logic of the program, decide on the desired output and the sequence or steps of operations to be executed. This is referred as design/development of an algorithm.

The general design/development of the algorithm are oriented primarily on the major activities and their relationships. This approach makes it easy to investigate alternative design approaches. Once a particular algorithm is finalized, a more detailed design incorporating both the major and minor activities and their relationships, calculations, data manipulations, logical operations, input and output can be completed.

Testing the logic of the problem on paper before coding is better to save time. If program codes are developed without testing the logic of the program a lot of unnecessary time has to be spent on back tracking to fix errors. Start coding when the logic of the program is correct in paper. Once the algorithm goes into the next step, it must be constantly revised to meet the changing needs of the company. A poorly designed algorithm is difficult to understand by teammates. Review program logic again and again until satisfied or ask teammates to evaluate the logic before proceeding to the next step.

Writing Program Code

Select a suitable programming language and write the program to implement the algorithm to get required solution to the problem. Begin programming by splitting the big program in to smaller blocks. Each block may represent each major step in the algorithm. The functions in the program must be independent of the main block. A function should be able to execute on its own as if it is a standalone block, but it can get the required input from the main block. Assemble each block in a bigger program and call them from the main block as the algorithm dictates. Initially, a program may contain errors but it is relatively easy to identify and correct them. Keep revising the program until a successful program is developed.

The following guidelines are useful in order to make program readable.

- Use comments in the program wherever necessary.
- Use meaningful identifiers.
- Use one line per statement.
- Use constants in all uppercase letters.
- Use indentation to elaborate the logical structure of a program.
- Use blank lines to separate function.

Some of the fundamental concepts of writing programs are,

- Programs are created to work on data and without data a program has nothing to do.
- Programs are run by computers and data are fed by programmer or user. Consequently, it is the program that should call the data when required via input statements and the program must process the input data and the results of processing the data are displayed to the user via output statements.
- Data becomes the major consideration for any program and a programmer needs to know the exact details about the format of the data read via the input statements.
- Programs written should be in general form, so that it can be extended for future use.
- Once the program starts its execution, it starts evaluating the first instruction, second instruction, and one after the other. The only time the statement doesn't execute in order, is when it executes a control statement. This may be inbuilt in the logic of the program. A programmer should be conscious enough in transferring the control from one part to another part of the program.

Debugging and Software Testing

The process of detecting and removing errors in a computer program is referred to as ***debugging***. Experience in working with computer programs has led to the observation that generally more than half of all programming effort and resources is spent in correcting errors in programs and carrying out modifications in programs.

Software testing by developers is constructive in nature and involves testing of individual modules and integration of modules. Testing by **I**ndependent **T**est **G**roup (ITG) leads to unbiased testing with the objective of finding errors. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding.

The software engineer creates a series of test cases that are intended to demolish the software that has been built. As a first step, start debugging the program, to identify and correct all errors that caused the program to produce incorrect results. Every programmer soon learns how easy it is to make mistakes.

The reason for errors is lack of time spent on planning, organizing and structuring the solution. When sufficient time is not spent on planning, the result is usually a structural mess of the logic of the problem. Test each function with a sample set of data for which the results are already known.

Use practical and realistic data as sample data. However, there is a possibility that user data may be non-relevant. This is what is called as data screening. This is usually done to get the required output. Catch each possible error and debug. Correct the syntactical errors if any pointed out by the compiler and execute the program again. Verify the results obtained and confirm that there are no logical errors in the program.

Two basic software testing techniques are,

- White box testing
- Black box testing

White Box Testing

White box testing is applied during the early stages of testing the software. It is used to verify the internal working of the program logic and with the help of tests conducted it is possible to ascertain whether the internal operation is performed as per the specification or not. Here, providing test cases that exercise specific sets of conditions and/or loops tests the logical paths through the software. Detailed testing is not possible. A limited number of important logical paths can be selected and exercised. The white box testing type verifies the following.

- Guarantees that independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true or false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Exercise internal data structures to assure their validity.

Black Box Testing

Knowing the specified blocks used in a program, tests can be conducted to demonstrate each block and its operations, simultaneously searching for errors. Black Box tests are used to demonstrate that the software blocks are operational, input is correctly accepted and output is produced correctly. Black box testing focuses on the functional requirements of the software and attempts to find errors in the following categories.

- Incorrect or missing blocks
- Interface errors
- Errors in data structures or data base access
- Performance errors
- Initialization/Termination errors

Even though a program works correctly on few data sets, it is not possible to ascertain that it will work properly on all data. The following two basic techniques are employed for testing and debugging programs on large data sets.

- Empirical testing
- Formal verification

In empirical testing, a large number of test cases are carefully selected and the program is run using the selected data set. If the program produces correct results for this collection of test data, then it can be ascertained that it will work properly on all set of data.

Formal verification is verifying the program mathematically and proving it. For each subprogram, there is a precise set of preconditions, which we assume is satisfied by the input parameters and a precise set of postconditions, which are satisfied by the output parameters. If any input satisfying the preconditions is always transformed to output satisfying the post conditions, then it is proved that the subprogram is correct. Selecting effective test data is a serious exercise. If the program is more significant, more care needs to be taken in the selection. When all the above steps are completed, the program can be released and set to work on live data.

Documentation

Documentation involves developing and writing supporting manuals and on-line assistance, which the user will need to understand and to use the finished program effectively. Documentation can be developed simultaneously when the program concepts are developed in the previous stages. The documentation details available in the previous stages are,

- The specification document
- The design document
- Algorithms for all sub programs/functions
- Programs and comments
- Sample test data

Now the job is to bring this material together into finished documents that clearly explain the users and other programmers how to use the program properly and effectively. It is not enough for a programmer to communicate with the computer but should also be able to communicate with the user and the teammates and this is achieved through documentation.

Program Maintenance

Programs are constantly revised to meet the changing needs of the company. Due to the cost involved in developing projects, programs are generally used for quite a long time. Program maintenance is the operation of adapting a program to keep it correct with changing specifications and new equipment. Each program along with its flowchart and other particulars are numbered and named along with the use of functions in the program. Preparation of data for the package and kept as a recorded report for easy retrieval for future use or modification of the program. Careful planning and designing of programs ensures that program maintenance is not a difficult task.

1.3 Software Design Techniques

Software/Program design is the utmost important step in writing any program. It is easy to write a computer program that is very simple. But practical software that has to be developed is at the best complex. When writing small program the importance of design is not readily visible. As the size of programs grew larger, the importance of design is evident.

The primary reason for the inclusion of this software design technique in the development of computer programs is to reduce the size and complexity of programs. It is a good idea to get into the habit of designing the programs from the very beginning, even though the design phase may seem like a waste of time. In order to solve larger problems, it is necessary to decompose the software into a number of small problems, which could be solved separately to obtain an integrated solution.

Benefits Of Program Design

Program designs are typically written in plain English, it is usually faster and easier to write the design than it is to write the code in a specific programming language. If an error is found in the logic in the design phase, it will be faster to fix the error than fixing the error after coding the algorithm. Sometimes, if program design is not carried carefully, thoughtfully, and completely, then after coding a large portion of program suffers from fundamental problems. This generally means that large portion of the code have to

be removed and start again from scratch. If program design is carefully analysed before coding then large portion of design have to be removed which cost much lesser time.

The process of designing a program allows understanding the incomplete specification of the original problem that may not be probably discovered until major portions of the program code are written. When the parts of the specification are confusing ask the client or teammates for clarification. A completed design also provides a means of comparing the specifications and the program function realistically before starting the coding phase. Any program worth writing must do what its specifications require and it must perform those tasks correctly and efficiently. Because a design in normal English is easier to understand the functions of the program at this phase than it is in coded form.

A good design will also allow program modification over time. Most programs are not fixed. That is, they change to incorporate new features, execute more efficiently, reduce complexity or overcome incorrect behavior.

A design breaks a large problem into multiple independent or semi-independent parts. These parts become modules that a programmer can reuse in other programs or replace with a similar but more efficient module or with a corrected module. Because the modules are small, they are relatively easy to understand and therefore easy to debug. Taken individually, they are simple and straightforward to implement.

1.4 Top-Down Design Approach

Top-down design is a disciplined method for designing the layout for anything that starts with the complete item then breaks it down into smaller and smaller subtasks. In programming, top down design means breaking the design of programs until the smallest component could be translated into an equivalent programming statement, till the smallest component is one instruction.

The idea behind top down design is to first divide the major work that a program will perform into a small number of broad subtasks.The top most level or the main module is known as level 0.

Whenever a subtask takes more than a very few lines of code to implement it, the process of subdividing is repeated. For each of the subtasks at level 0, start a new module at level 1. Each of these subtasks is considered individually and again divided into several sub-subtasks. The process of subdividing and creating new subtasks or levels is continued until each part of the problem is easily translatable into a particular programming language in one or a very few lines of code and thus do not need further expansion.

The result of the top-down program design is a “tree-like” structure. Just as the trunk of a tree divides into several large branches, each large branch divides into smaller branches and each small branch divides into twigs, the modules at each level of a top down design spawns several modules at the next level down and each of these gives rise to several more modules at the next level. Fig. 1.2 shows a three level top-design approach.

Top-down program design may extend to several levels depending upon the problem to be solved. The decomposition and simplification task is performed again and again on the original problem and then on the successive sub functions, until finally left with the task that is so elementary and need not to be simplified further.

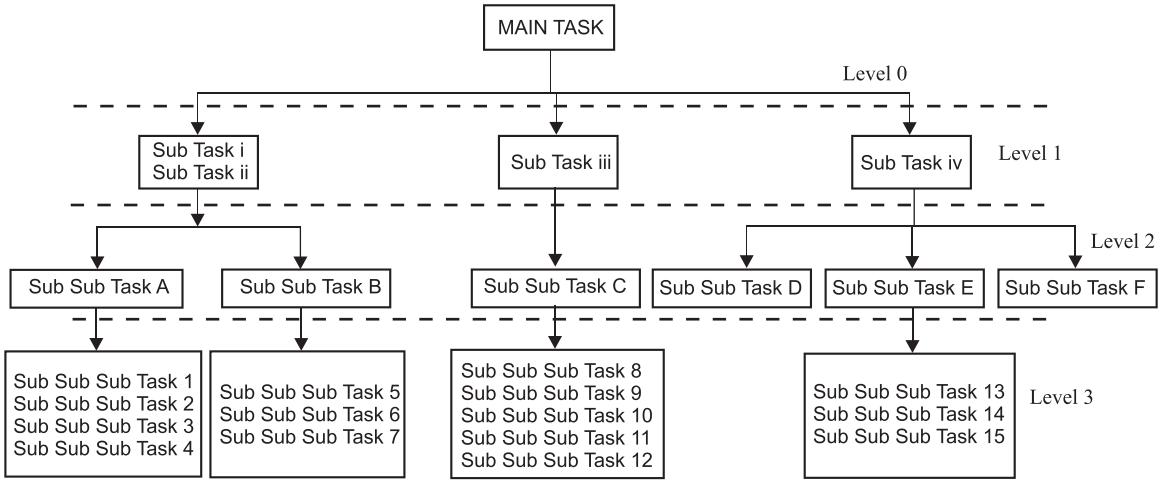


Fig. 1.2: *Top-down design approach.*

Steps in Top-Down Design

- Problem decomposition
  - o Divide the whole job into smaller tasks and give each task a name that represents work done by the task and list all these tasks in their order of execution.
  - o Then divide each of the smaller tasks into smaller sub-task in the same way and list the smaller tasks underneath the task to which they are part of.
  - o Stop when a task is so simple, that need not be split it up any more.
- Collecting information required for each task
  - o For each task, list the input information the task requires to do its work. Also list the information it needs to produce as output.
  - o List the need for each task and the result it should produce.
  - o Remember that a task is succeeded or not, since it counts as information it must produce.
- Determining data representation
  - o Decide on type of data and data structures for each kind of information that the task requires to carry its work.
- Writing functions declarations
  - o Most tasks at each level are going to be functions. Decide on the arguments needed, for each function and the type of value it returns.

1.5 Programming Methodologies

Programming by a good programmer is something similar to an art. Unfortunately, it is quite common to develop programs that look meaningless and programs do not give expected results. Many brilliant minds have worked to devise methodologies to discipline the programmer to create programs that are useful and maintainable. Programming methodologies is a complex field, with many methodologies, names, many goals and means to reach them.

Some of the important programming methodologies are,

- Stepwise refinement
- Modularity
- Structured programming

Stepwise Refinement

Stepwise refinement is a method to decompose a problem into smaller parts. The problem can be easily solved if the problem is split in to smaller steps. The best way to code complicated programs is to construct the hierarchy one level at a time, finally writing the actual steps when the task is small enough to be easily done. Splitting the program and analysing each part of the program for correctness before continuing to the next part of the program improves the efficiency of the program.

A program is gradually developed in a sequence of refinement steps. In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and therefore be guided by the facilities available on that computer language.

The result of the execution of a program is expressed in terms of data and it may be necessary to introduce further data for communication between the obtained subtasks or instructions. As tasks are refined, the data may have to be refined, decomposed, or structured and it is natural to define program and data specifications in parallel. Every refinement step implies some design decisions. It is important that these decisions be made explicit and that the programmer should be aware of the underlying criteria and of the existence of alternative solutions.

The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of decision. Sub-trees may be considered as families of solutions with certain common characteristics and structures. The notion of such a tree may be particularly helpful in the situation of changing purpose and environment to which a program may sometime have to be adapted.

Modularity

Modularity is the process of splitting a large program/project into smaller modules to perform the operations fast which helps in easy error checking. Modules should be designed, implemented, and documented with a regard to their possible future use in other projects. Modularity is essential for sound software design. Modular programs are easier to develop and test, especially for a team of programmers. They are also easier to understand and maintain because certain changes can be implemented locally and do not require extensive modifications or re-testing the entire application.

Large projects may involve hierarchies of modules. One approach to good software architecture is to arrange functions in layers based on their level of functionality and the level of data structures that they deal with. Each layer can be implemented as a separate module or several modules. In an ideal architecture, each layer uses only functions and data structures from the layer immediately below it. Changes in the implementation of a module normally do not disturb other modules and hence saves work and time.

Modularity in design is easy to describe and complex to implement. The benefits of modular software design, besides cleaner structure and more efficient implementation, include easier software maintenance and reusability. In a project, each module carries out an independent task. Once the interfaces between the modules are defined, each module can be implemented and even modified independently of the other modules. This property is referred as ***locality***. A tested module can be used in other projects that present the same task. This property is referred as ***reusability***.

The process of combining different object modules into one executable module is referred as ***linking***. The modules are compiled separately and linked together into one executable program by a linker program. Object modules may be combined into object module libraries by using a library utility program. The linker can search specified libraries for modules that supply remaining unresolved external references and include them into the executable file.

A special program, the linker, examines the set of object modules that are specified in the linking command. It combines all the code contained in them and attempts to resolve all external references by finding their names and addresses among the global of other modules. The logical addresses of externals are then replaced with their actual addresses in the combined code, and the linker creates an executable file.

Modern programming languages and software development environments, including C, provide support for this modular approach. Each module is usually implemented in two separate files, a header file, and a source file. The header file may contain constants, function definitions, definitions of data structures and declarations of external variables. The source code contains static variables, function call statements (for the functions defined in the header file, with some other additional functions defined in it) and its own functions.

1.6 Problem Analysis Chart (PAC)

A ***Problem Analysis Chart*** (PAC) is a structured pre-programming tool used to break down a problem to be programmed into key components. PAC helps to systematically understand the problem, identify necessary input and output data and develop a structured approach to solve the problem. PACs serve as the initial step in problem-solving by organising information that will be used to visualise and implement the solution.

A PAC is divided into four main sections:

1. **Given Data (Inputs):** All data provided by the user or an external source.
2. **Required Results (Outputs):** The final information or results the program must produce.
3. **Processing Required:** The equations, logic, loops, and conditions needed to transform inputs into outputs.
4. **Solution Alternatives:** Alternative approaches, constraint handling, or unique data validation paths.

EXAMPLE 1.1 :

Develop a PAC (Program Analysis Chart) to compute area and circumference of a circle of radius, r.

PAC to calculate area and circumference of a circle of radius r

1. Given Data (Inputs)	**2. Required Results (Outputs)**
• Radius of circle, r	• Area of circle, A • Circumference of circle, C
3. Processing Required	**4. Solution Alternatives**
• A = 3.142 * r * r • C = 2 * 3.142 * r	• Can define 3.142 as macro, pi • Can reject negative r input using some control statement

EXAMPLE 1.2 :

Develop a PAC (Program Analysis Chart) to determine largest of two numbers.

PAC to determine largest of two numbers

1. Given Data (Inputs)	**2. Required Results (Outputs)**
• First number, Numb1 • Second number, Numb2	• Largest of Numb1, Numb2
3. Processing Required	**4. Solution Alternatives**
• Compare numbers using relational operator	• Can use if-else control structure

EXAMPLE 1.3 :

Develop a PAC (Program Analysis Chart) to compute factorial of a number.

PAC to compute factorial of a number

1. Given Data (Inputs)	**2. Required Results (Outputs)**
• Integer number, n	• Factorial of n
3. Processing Required	**4. Solution Alternatives**
• Initialize, factorial = 1 • Decrement, n, check n is 0 • Till n is 0, factorial = factorial * n	• Can reject negative n input using some control statement

1.7 Flowchart

Flowchart is a graphical representation of the operation flow of a program. It is also the graphical form of an algorithm. Flowchart can be a valuable aid in visualizing a program. The various symbols used for drawing flowcharts are shown in Fig. 1.3. A sample flowchart is shown in Fig. 1.4. The operations represented by various symbols of flowchart are explained in Table 1.1.

Rules for Drawing Flowcharts

- A flowchart should start and end with terminal symbols.
- Line and arrow should be used in between other symbols to indicate the flow of the flowchart.
- The logic of the flowchart should flow from top to bottom or from left to right.
- Instructions inside the flowchart symbols should be independent of programming languages.
- To avoid cross over in flowchart use connectors (on-page connector).
- If the flowchart is large then make use of off-page connector to extend the flowchart to next page.

Guidelines for Drawing Flowcharts

- Always include a title for the flowchart.
- Always number the steps in the process of the flowchart.
- Always include a key for symbols for those who are unfamiliar with flowcharts.
- Draw the flowchart from top to bottom or left to right so that the flowchart is consistent.
- Create the flowchart beginning with general operations on the first draft.
- Do not be concerned about defining the process specifically on the first draft.
- Review flowchart for proper flow of information and sequence of operations.
- Flowchart should reflects all major decisions.
- It is possible to perform a highly detailed analysis of any process using the flowchart.
- Be sure to involve teammates who are involved in the process to get accurate perceptions.

Note : *Line and arrow are also called sequence symbols.*

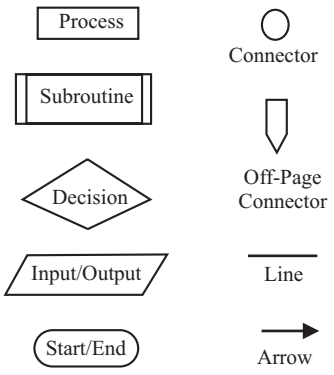


Fig. 1.3: *Symbols used in a flowchart.*

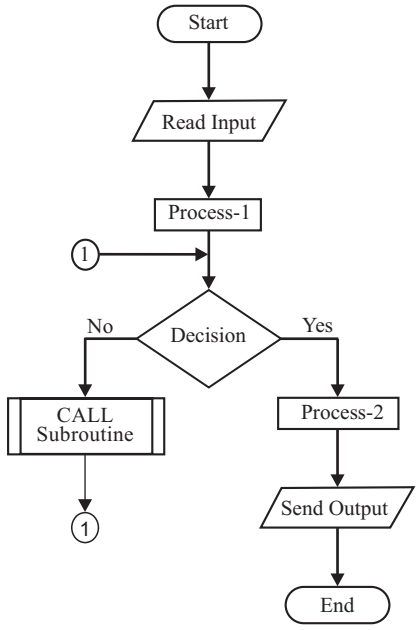
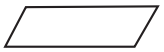
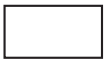
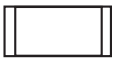
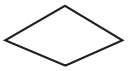


Fig. 1.4: *A sample flowchart.*

Table 1.1: Operations Represented by the Symbols used in Flowchart

Symbol	Operations
Racetrack shape box 	A racetrack shaped symbol is used to indicate the begining (start) or end of a program.
Parallelogram 	A parallelogram is used to represent input or output operation.
Rectangular box 	A rectangular box is used to represent simple operations other than input and output operations.
A rectangular box with double lines on vertical sides. 	A rectangular box with double lines on vertical sides is used to represent a subroutine or procedure.
Diamond shaped box 	A diamond shaped box is used to represent a decision point or cross road in the programs.
Small circle 	A small circle is used as a connector to show the connections between various parts of a flowchart within a page. Identical numbers are entered inside the circles that represent the same connecting points.
Five-sided box 	A five-sided box symbol is used as an off-page connector to show the connections between various sections of a flowchart in different pages. Identical numbers are entered inside the boxes that represent the same connecting point.
Line	Lines are drawn between boxes and diamonds to indicate the program flow.
Arrow	Arrow are placed on the lines to indicate the direction of program flow.

EXAMPLE 1.4 :

Draw a flowchart to find the sum of two numbers.

FLOWCHART :

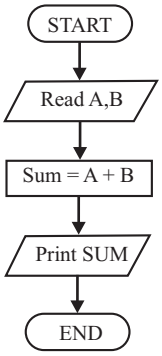


Fig. 1: *Flowchart to find the sum of two numbers.*

EXAMPLE 1.5 :

Draw a flowchart to find the biggest of two numbers.

FLOWCHART :

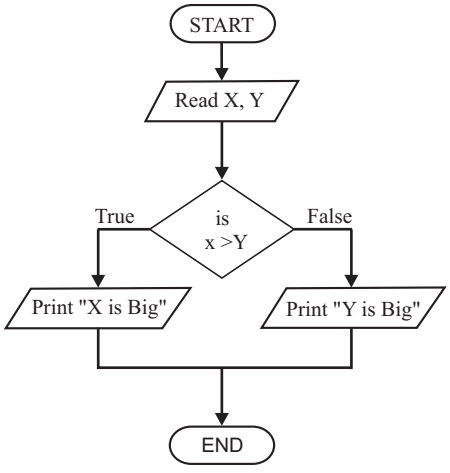


Fig. 1: *Flowchart to find the sum of two numbers.*

EXAMPLE 1.6 :

Draw a flowchart to find the sum and average of N numbers.

FLOWCHART :

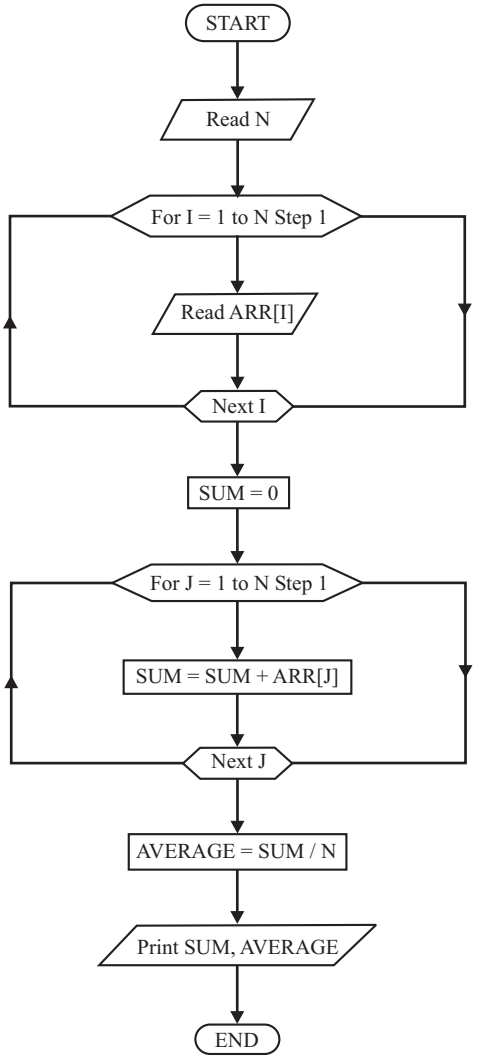


Fig. 1: *Flowchart to find the sum and average of numbers.*

EXAMPLE 1.7 :

Draw a flowchart to find the sum of the series $1 + X + X^2 + X^3 + \ldots\ldots$

FLOWCHART :

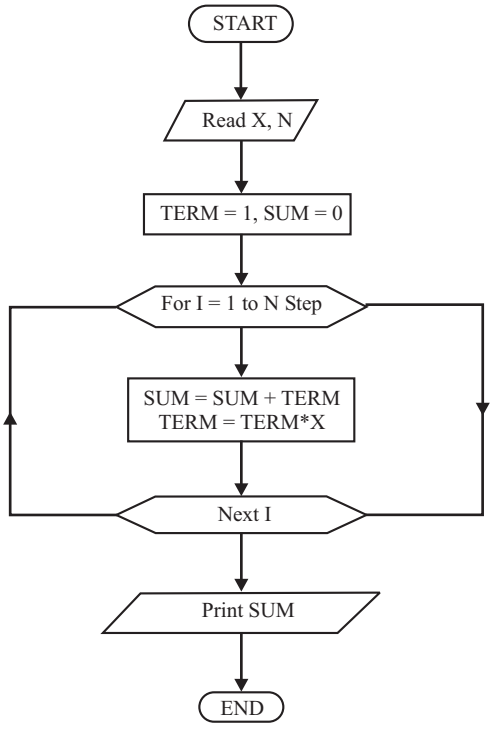


Fig. 1: *Flowchart to find the sum of the series.*

EXAMPLE 1.8 :

Draw a flowchart to find the factorial of a number using functions.

FLOWCHART :

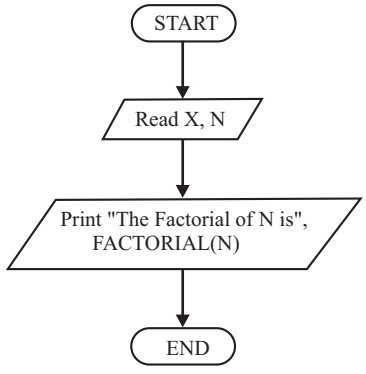


Fig. 1: *Flowchart to find the factorial of a number.*

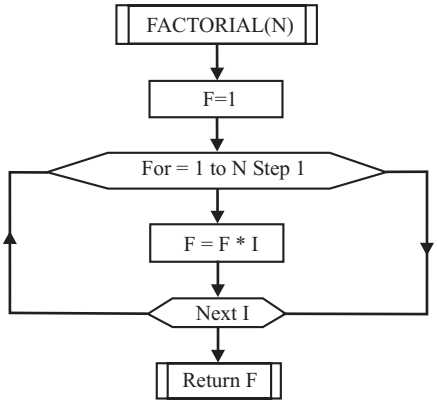


Fig. 2: *Flowchart for factorial function.*

EXAMPLE 1.9 :

Draw a flowchart to find the roots of the quadratic equation.

FLOWCHART :

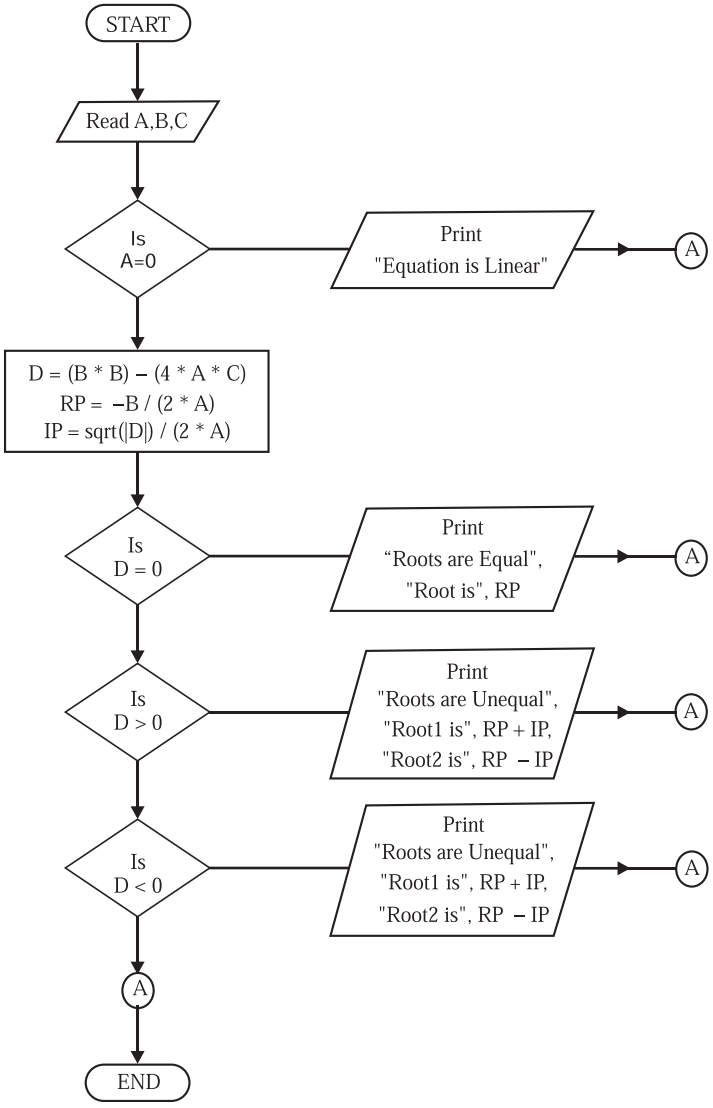


Fig. 1: *Flowchart to find the roots of the quadratic equation.*

1.8 Pseudocode

Pseudocode is a step-by-step, plain-English description of a computer program or algorithm designed for human reading. Pseudocode cannot be compiled or run on a computer.

> ***Note:*** *Pseudocode is pronounced as soo-doh-code.*

Pseudocode is entirely syntax-free and it serves as an informal blueprint to help programmers map out logic and communicate ideas clearly before writing actual code in any computer programming language. Each textbook and individual software developer will have their own personal style of pseudocode. There is no universal standard for pseudocode.

Advantages of pseudocode

- Eliminates the difficulty of semi-colons, brackets and language-specific rules.
- Allows technical and non-technical persons to review the workflow easily.
- Help to eliminate logical errors and flaws early in the design phase.
- Can be translated into any target programming language.

EXAMPLE 1.10 :

Write pseudocode to find sum and average of two numbers.

Pseudocode to find sum and average of two numbers

```
START
    DECLARE numb1, numb2, sum, average AS FLOAT
    PRINT "Enter two numbers:"
    INPUT numb1, numb2
    COMPUTE, sum = numb1 + numb2
    COMPUTE, average = sum / 2
    PRINT "Sum =:", sum
    PRINT "Average =:", average
END
```

EXAMPLE 1.11 :

Write pseudocode to find a number is even or odd number.

Pseudocode to find a number is even or odd number

```
START
    DECLARE number AS INTEGER
    PRINT "Enter an integer:"
    INPUT intnumber
    IF
    intnumber MOD 2 == 0
    THEN
    PRINT "The number is even."
    ELSE
    PRINT "The number is odd."
    ENDIF
END
```

EXAMPLE 1.12:

Write a pseudocode to find factorial of a number.

Pseudocode to find factorial of a number

```
START
    DECLARE number, factorial, i AS INTEGER
    SET, factorial = 1
    PRINT "Enter a positive integer:"
    INPUT intnumb
    IF
    intnumb < 0
    THEN
    PRINT "Error: Factorial not defined for negative numbers."
    ELSE
    FOR i = 1 TO intnumb DO
    factorial = factorial * i
    ENDFOR
    PRINT "Factorial is:", factorial
    ENDIF
END
```

1.9 Algorithm

An ***algorithm*** is a step-by-step recipe for solving an instance of a problem. Every single procedure that a computer performs is an algorithm. In this way, every step a computer takes is the result of a single instruction in an algorithm. An algorithm is a precise procedure for solving a problem in finite number of steps. An algorithm states the actions to be executed and the order in which these actions are to be executed.

An ***algorithm*** is a well-ordered collection of clear and simple instructions of definite and effectively computable operations that, when executed, produces a result and stops executing at some point in a finite amount of time rather than just going on and on infinitely.

The various steps in developing algorithms are,

- Devising the algorithm
- Validating the algorithm
- Expressing the algorithm

Devising the algorithm is a method for solving a problem. Each step of an algorithm must be precisely defined and no vague statements should be used. Pseudocode is used to describe the algorithm, in less formal language than a programming language. The next step in developing algorithm is validating the algorithm (i.e., the proof of correctness of the algorithm). A programmer must be able to perform each step using paper and pencil by giving the required input, use the algorithm and should get the required output in a finite amount of time. The next step is to implement the algorithm in a programming language. The algorithm used should terminate after a finite number of steps.

> ***Note: Pseudocode*** *is a high-level, informal description of a computer program or algorithm. Pseudocode uses a mix of English and simple programming structures to outline logic without considering the strict syntax rules of a specific programming language.*

Properties of Algorithm

- An algorithm takes zero or more inputs.
- An algorithm results in one or more outputs.
- All operations can be carried out in a finite time.
- An algorithm should be efficient and flexible.
- It should use less memory space as much as possible.
- An algorithm must terminate after a finite number of steps.
- Each step in the algorithm must be easily understood for someone reading it.
- An algorithm should be concise and compact to facilitate verification.

Relation between PAC, Flowchart, Pseudocode and Algorithm

PAC, flowchart, pseudocode and algorithm are interconnected in problem-solving using computer programming. By integrating PAC, flowchart, pseudocode and algorithm, solution for problems using computer programming can be systematically analysed, visualise the solution process and implement efficient solutions.

This structured approach enhances programmer problem-solving skills and ensures that solutions are well-organized, efficient and easy to understand.

- PAC provides a structured approach to analyse the problem by identifying inputs, outputs, processes and alternative solutions.
- Flowchart visually represents the steps and decision points outlined in the PAC, making it easier to understand and communicate the solution process.
- Pseudocode provides logical description in natural language (English) for better understanding of sequence of operations.
- Algorithm provides exact logical sequence of operations to solve the problem in any computer programming language.

EXAMPLE 1.13 :

Write an algorithm to find the sum of two numbers.

Algorithm to find the sum of two numbers

STEP 1 : Read the input values for A and B.

STEP 2 : Compute, S = A + B.

STEP 3 : Print "The Sum of A+B is", S.

STEP 4 : Stop.

EXAMPLE 1.14 :

Write an algorithm to find the biggest of two numbers.

Algorithm to find the biggest of two numbers

STEP 1 : Read the input values for X and Y.

STEP 2 : If X > Y.

Print "X is Big".

else

Print "Y is Big".

[End of If Structure]

STEP 3 : Stop.

EXAMPLE 1.15 :

Write an algorithm to find the biggest of three numbers.

Algorithm to find the biggest of three numbers

STEP 1 : Read the input values for 3 numbers as A, B and C.

STEP 2 : Check whether A is greater than B or not.

STEP 3 : If **STEP 2** is True, Check whether A is greater than C or not.

STEP 4 : If **STEP 3** is True, Print "A is the Biggest number". Goto **STEP 10.**

STEP 5 : If **STEP 2** is False, Check whether B is greater than C or not.

STEP 6 : If **STEP 5** is True, Print "B is the Biggest number". Goto **STEP 10.**

STEP 7 : If **STEP 3** is False, Goto **STEP 9**.

STEP 8 : If **STEP 6** is False, Goto **STEP 9**.

STEP 9 : Print "C is the Biggest number". Goto **STEP 10.**

STEP 10 : End.

EXAMPLE 1.16 :

Write an algorithm to find the roots of the quadratic equation.

Algorithm to find the roots of the quadratic equation

STEP 1 : Read the input as numbers as A, B and C.

STEP 2 : Check whether A is equal to 0 or not.

STEP 3 : If **Step 2** is True,

Print "Equation is Linear".

Goto **Step 11.**

STEP 4 : If **Step 2** is False, Evaluate D = B2 − 4AC and REAL = −B / (2A)

STEP 5 : Check whether D is equal to 0 or not.

STEP 6 : If **Step 5** is True,

Print, "Roots are Real and Equal". Print REAL. Goto **Step 11.**

STEP 7 : If **Step 5** is False, Check whether D is greater than 0 or not.

STEP 8 : If **Step 7** is True,

Evaluate R1 = (−B + sqrt(D)) / 2A) and R2=(−B +sqrt(D)) / 2A)

Print "Roots are Real and Unequal". Print R1 and R2.

Goto **Step 11.**

STEP 9 : If **Step 7** is False, Check whether D is lesser than 0 or not.

STEP 10 : If **Step 9** is True,

Evaluate D = −D, IMAG = SQRT (D) / (2A)

R1 = REAL + IMAG and R2 = REAL = IMAG

Print "Roots are Imaginary". Print R1 and R2.

STEP 11 : End.

EXAMPLE 1.17:

Write an algorithm to find the sum of the series $1 + X + X^2 + X^3 + \ldots$ using functions.

Algorithm to find the sum of the series

STEP 1 : Read the input values as numbers for X and the limit for N.

STEP 2 : Print "Sum of Series is ", SUM_SERIES(X, N).

STEP 3 : End.

SUM_SERIES(X, N)

X : REAL, N : Integer

STEP 1 : Initialize, TERM = 1.0, SUM = 0.0

STEP 2 : Repeat, For I = 1 to N.

STEP 3 : SUM = SUM + TERM.

STEP 4 : TERM = TERM * X.

STEP 5 : Increment I by 1.

Step 6 : [End of **STEP 2** For loop].

STEP 7 : Return SUM.

END SUM_SERIES()

EXAMPLE 1.18:

Write an algorithm to find the factorial of a number using recursion.

Algorithm to find the factorial of a number using recursion

STEP 1 : Read the input value as number for N.

STEP 2 : Print "The Factorial of ", N, "is ", FACT(N).

STEP 3 : End.

FACT(N)

N : INTEGER

STEP 1 : Check whether N is lesser than or equal to 1 or not.

STEP 2 : If **STEP 1** is True, Return 1.

STEP 3 : If **STEP 1** is False, Return (N * FACT(N − 1)).

END FACT()

EXAMPLE 1.19:

Write an algorithm to sort the numbers in an array in ascending order.

Algorithm to sort the numbers in an array in ascending order

STEP 1 : Read the input value as a number for limit N.
STEP 2 : Repeat, For I = 0 to N.
STEP 3 : Read A[i].
STEP 4 : Increment I by 1.
STEP 5 : [End of **STEP 2** For loop].
STEP 6 : Repeat, For I = 0 to N−1.
STEP 7 : Repeat, For J = I to N.
STEP 8 : Check whether A[I] is greater than A[J] or not.
STEP 9 : If **STEP 8** is True, Swap A[I] and A[J].
STEP 10 : Increment J by 1.
STEP 11 : [End of **STEP 7** For loop].
STEP 12 : Increment I by 1.
STEP 13 : [End of **STEP 6** For loop].
STEP 14 : Repeat, For I = 0 to N.
STEP 15 : Print A[i].
STEP 16 : Increment I by 1.
STEP 17 : [End of **STEP 14** For loop].
STEP 18 : End.

EXAMPLE 1.20:

Write an algorithm to sort the strings in alphabetical order.

Algorithm to sort the strings in alphabetical order

STEP 1 : Read the input value as a number for limit N.
STEP 2 : Repeat, For I = 0 to N.
STEP 3 : Read STR[i].
STEP 4 : Increment I by 1.
STEP 5 : [End of **STEP 2** For loop].
STEP 6 : Repeat, For I = 0 to N−1.
STEP 7 : Repeat, For J = I to N.
STEP 8 : Check whether STR[I] is greater than STR[J] or not.
STEP 9 : If **STEP 8** is True, Swap STR[I] and STR[J].
STEP 10 : Increment J by 1.
STEP 11 : [End of **STEP 7** For loop].
STEP 12 : Increment I by 1.
STEP 13 : [End of **STEP 6** For loop].

STEP 14 : Repeat, For I = 0 to N.

STEP 15 : Print STR[i].

STEP 16 : Increment I by 1.

STEP 17 : [End of **STEP 14** For loop].

STEP 18 : End.

EXAMPLE 1.21:

Write an algorithm to find the length of a string.

Algorithm to find the length of the string

STEP 1 : Read the input string STR.

STEP 2 : Repeat, For STR[I] = 0 to '\0' (i.e., NULL character)

STEP 3 : Increment I by 1.

STEP 4 : [End of **STEP 2** For loop].

STEP 5 : Print "The number of characters in the string is ", I.

STEP 6 : End.

EXAMPLE 1.22:

Write an algorithm to check whether the given string is palindrome or not.

Algorithm to check whether the given string is palindrome or not

STEP 1 : Read the input string STR.

STEP 2 : Repeat, For STR[I] = 0 to '\0' (i.e., NULL character)

STEP 3 : Increment I by 1.

STEP 4 : [End of **STEP 2** For loop].

STEP 5 : Repeat, For K = 0 to i, J = i − 1.

STEP 6 : Check whether STR[K] is not equal to STR[J] or not.

STEP 7 : If **STEP 7** is True, FLAG = 1.

STEP 8 : Increment K by 1.

STEP 9 : Decrement J by 1.

STEP 10 : [End of **STEP 5** For loop].

STEP 12 : Check whether FLAG is equal to 1 or not.

STEP 13 : If **STEP 12** is True, Print "The string is not a palindrome"

STEP 14 : If **STEP 12** is False, Print "The string is a palindrome"

STEP 15 : End.

1.10 Loop Constructs

Loop constructs are fundamental programming structure that allows a specific block of code to be executed repeatedly, either for a predetermined number of times or until a particular condition is met. Loops are essential for automating repetitive tasks, reducing code duplication and improving the overall efficiency of a program.

The loop constructs consist of following primary components.

- **Initialization:** Setting up a loop control variable (e.g., a counter) to its starting value.
- **Condition/Test:** A logical expression checked before or after the each iteration. The loop continues only as long as logical expression is true.
- **Update/Iteration Step:** Modifying the control variable (e.g., incrementing or decrementing) to move toward the termination condition.
- **Loop Body:** The actual set of instructions or code block that is repeated in the every iteration.

The popular loop constructs are,

- sequence
- if-then-else
- for
- while
- repeat-until
- case

Each of above constructs can be embedded inside any other construct. The constructs represent the logic, or flow of control in an algorithm. These constructs are sufficient to implement the flow of control in any algorithm.

Sequence Construct

Sequence construct is a linear progression where one task is performed sequentially one after the other. Each action is written in a single line and all actions are aligned with the same indent. They are written in the sequence (top to bottom) in which the actions are to be performed. For example, for finding the area of a rectangle, the following sequence is employed.

`READ` height of rectangle

`READ` breadth of rectangle

`CALCULATE` area by multiplying height breadth

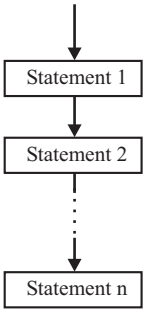


Fig. 1.5: *Sequence construct.*

If-then-else Construct

If-then-else construct is a decision-making statement in which a choice is made between two alternative courses of action. The if-then-else construct uses four keywords. They are, ***if, then, else*** and ***endif***.

General form of usage of the if-then-else construct is,

```
if condition then
sequence-1
else
sequence-2
endif
```

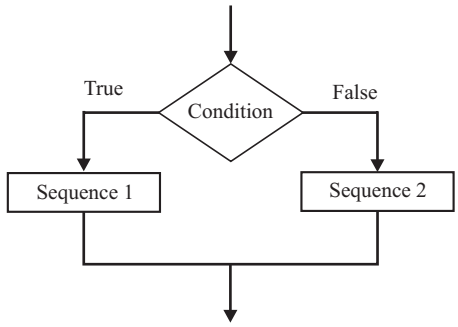


Fig. 1.6: *If-then-else construct.*

The if-then-else construct checks for a condition at its beginning. If the condition is true, it will execute `sequence-1`, otherwise it will execute `sequence-2`.

The else keyword and `sequence-2` are optional and are used depending upon the problem. For example, the following code helps to find the greater of two numbers.

```
VAR A, B
IF A > B THEN
Display "A is Big"
ELSE
Display "B is Big"
ENDIF
```

For Construct

For construct is a special loop in which an index variable is automatically initialized, incremented, and tested. The beginning and ending of the loop are indicated by two keyword ***for*** and ***endfor***.

The general form of usage of the for construct is,

```
for index = start TO finish BY incr DO
SEQUENCE(S)
endfor
```

The loop is executed and the `SEQUENCE` is performed only if the condition is true. At the end of the iteration, the condition is checked and the loop continues as long as the condition is true. At the initiation of the loop, the index variable is set to the `start` value. At the end of the each iteration, the index variable is automatically incremented by the DO value. The loop repeats until the index value reaches the `finish` value. For example, the following code helps to write a program to display `first 10` numbers in the output.

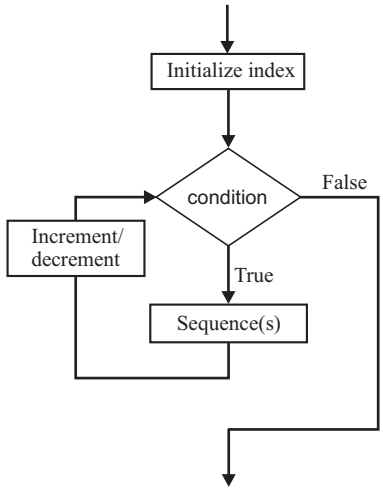


Fig. 1.7: *For construct.*

```
for I = 1 TO 10 BY incr 1
DISPLAY I
Endfor
```

While Construct

While construct is a loop construct with a conditional check in the beginning. The beginning and ending of the loop are indicated by two keywords ***while*** and ***endwhile***.

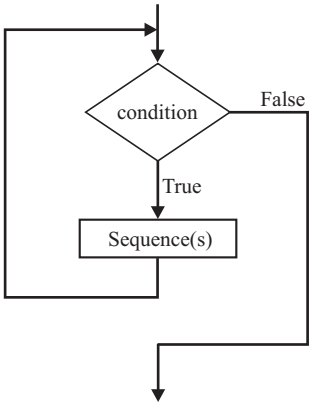


Fig. 1.8: *While construct.*

The general form of usage of the while construct is,

```
while condition
SEQUENCE(S)
endwhile
```

The `SEQUENCE` is executed only if the condition is true. At the end of the each iteration, the condition is checked and the loop continues as long as the condition is true. For example, the following code helps to write a program to count the number of digits in an integer.

```
VAR number, digit
while number  > 0
increment digit by 1
number = number / 10
endwhile
```

Repeat-until Construct

Repeat-until is a loop, similar to the while loop except that the conditional check is performed at the end of the loop instead of beginning of the loop. The beginning and ending of the loop are indicated by two keyword s ***repeat*** and ***until***.

General form of usage of the repeat-until construct is,

```
repeat
SEQUENCE(s)
until condition
```

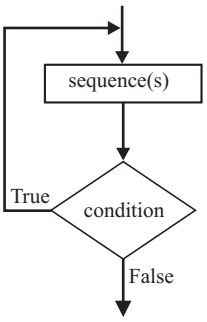


Fig. 1.9: *Repeat-until construct.*

The `SEQUENCE` in this type of loop is always performed at least once (even when the condition is false), because the test is performed after the `SEQUENCE` is executed. At the end of the iteration, the condition is checked, and the loop repeats until the condition is true. The loop terminates when the condition becomes false. For example, the following code helps to write a program to count the number of digits in an integer.

```
VAR number, digit
repeat
increment digit by 1
number = number / 10
until number > 0
```

Case Construct

Case construct provides a multi-way branch statement based on the value of an expression. Case is a generalization of the if-then-else construct. The case construct uses four keywords. They are ***case, of, others*** and ***endcase***.

The general form of usage of the case construct is,

```
case expression of
condition 1 : sequence 1
condition 2 : sequence 2
condition 3 : sequence 3
... ...
... ...
... ...
... ...
condition n-1 : sequence n-1
condition n : sequence n
others :
default sequence
endcase
```

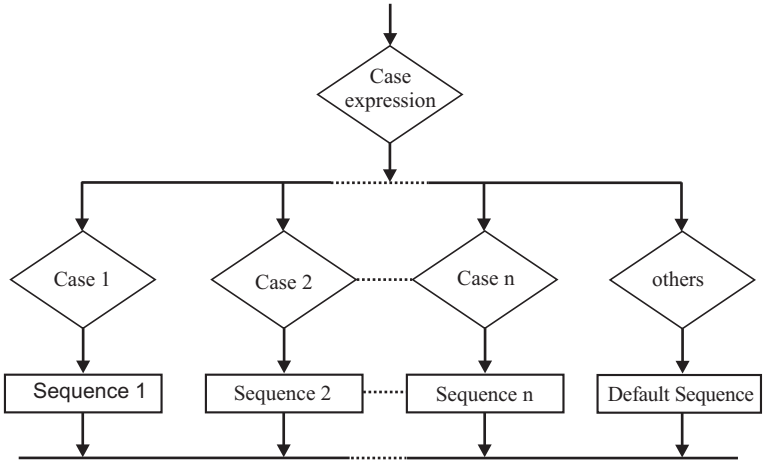


Fig. 1.10: *Case construct.*

The others clause with its default sequence is optional. Conditions are normally numbers or characters indicating the value of expression, but they can be English statements or some other notation that specifies the condition under which the given sequence is to be performed. The sequence may be associated with more than one condition.

For example, the following code helps to write a program to display the grades of the students in their exams.

```
case grade of
40 : print "Fail"
50 : print "Pass"
55 : print "Pass with Second Class"
60 : print "Pass with First Class"
75 : print "Pass with Distinction"
Endcase
```

1.11 Summary of Important Concepts

1. A computer program is a series of instructions to carry out a particular task written in a language that a computer can understand. The process of preparing and feeding the instructions into the computer for execution is referred as programming.
2. The steps adopted to develop a program in any programming language includes, problem definition, collection of information, selection of solution, development of algorithm, writing program code, debugging and testing, documentation and program maintenance.
3. The process of detecting and removing errors in a computer program is referred to as debugging.
4. Software testing is a critical element of software quality assurance and performs the ultimate review of specification, design, and coding.
5. White box testing is used to verify the internal working of the product, with the help of tests conducted to ascertain whether the internal operation is performed as per the specification or not.
6. Black Box tests are used to demonstrate that the software blocks are operational, input is correctly accepted, and output is produced correctly or not.
7. Two basic techniques for debugging programs are empirical testing and formal verification.
8. Documentation involves developing and writing supporting manuals and on-line assistance, which the user need to understand and use the finished program effectively.
9. Top-down design is a disciplined method for designing the layout for anything that starts with the complete item then breaks it down into smaller and smaller subtasks.
10. Stepwise refinement is a method to decompose a problem into smaller parts i.e., dividing the problem into smaller steps so that programmer easily solve the problem.
11. Modularity is the process of splitting a large program / project in to smaller modules to perform the operations fast and, which helps in easy error checking.
12. Once the interfaces between the modules are defined, each module can be implemented and even modified independently of the other modules such property is referred to as locality.
13. A tested module can be used in other projects that present the same task, such property is referred as reusability.
14. The process of combining different object modules into one executable module is referred as linking. Selection is the process of selecting alternative steps that may be taken subject to solve a particular condition.

15. Iteration is the process of repeating certain steps until a particular condition is true. This is achieved in programming by using loops
16. An algorithm is a well-ordered collection of clear and simple instructions of definite and effectively computable operations that, when executed, produces a result and stops executing at some point in a finite amount of time.
17. The various steps in writing algorithms are, devising the algorithm, validating the algorithm and expressing the algorithm.
18. Flowchart is a graphical representation of the operation flow of a program. It is also the graphical form of an algorithm. Flowcharts can be a valuable aid in visualizing programs.