Second Edition

Programming in C

FIREWAL MEDIA

J.B. DIXIT

Published by :

FIREWALL MEDIA

(An Imprint of Laxmi Publications Pvt. Ltd.)

113, Golden House, Daryaganj,
New Delhi-110002

Phone : 011-43 53 25 00
Fax : 011-43 53 25 28

www.laxmipublications.com
info@laxmipublications.com

Price : **Rs. 350.00** *Only.*

Edited by : Sangeeta Dixit

First Edition : 2005
Second Edition : 2009

OFFICES

✆ **Bangalore**	080-26 61 15 61	✆ **Chennai**	044-24 34 47 26
✆ **Cochin**	0484-239 70 04	✆ **Guwahati**	0361-254 36, 69, 251 38 81
✆ **Hyderabad**	040-24 65 23 33	✆ **Jalandhar**	0181-222 12 72
✆ **Kolkata**	033-22 27 43 84	✆ **Lucknow**	0522-220 95 78
✆ **Mumbai**	022-24 91 54 15, 24 92 78 69	✆ **Ranchi**	0651-221 47 64

FPR-3027-350-PROGRAMMING IN C

C. 235 /09/06

Typeset at : ABRO Enterprises, Delhi.

Printed at : Ajit Printers, Delhi.

CONTENTS

ACKNOWLEDGEMENT

It is with a great sense of satisfaction that I acknowledge the help and support rendered to me by many people in bringing this book in its current form.

My heartiest thanks to Mr. Raman Sharma (Manager—WIPRO) and Mrs. Ruchi Sharma (M.C.A., M.Phil) for their creative and thoughtful association in the preparation of this book.

My lovely children Apoorva, Aanchal and Vansh always remind me about the work to be completed with their ever smiling faces. So, a special thank to them also.

I would like to thank all my teachers, members of my family, students and well wishers whose blessing, knowledge, advice and interaction have made this project a possible venture in my life.

—AUTHOR

PREFACE TO THE SECOND EDITION

It is a matter of great pleasure for me to write the preface for the second edition of this book. The most comprehensive, authoritative, and up-to-date book on C programming language. It provides a unique combination of programming and problem solving through 'C' language.

This book is written for M.C.A., M.Tech., M.Sc., Engineering, B.C.A., B.I.T., B.Sc., C-DAC, DOEACC O-Level and A-Level, P.G.D.C.A. and other computer programme's students. In addition it can be of great help to those who feel that programming is not their cup of tea. I am sure that the novice will feel quite comfortable after going through the subject matter due to its *practicality, readability* and *correctness.* All the programs in this book have been tested on Turbo C compiler. This book is organized into fourteen chapters as per the contents. A brief overview of all the chapters is presented below :

Chapter 1: Introduction to Problem Solving

It introduces the meaning of algorithms, flowcharts, decision tables and their need. The various problem solving methodologies and programming languages are also given.

Chapter 2: Overview of C

It includes character set, C tokens, keywords and identifiers. Structure of a C program and its execution are also given. Constants, variables and data types are also explained here.

Chapter 3: Operators and Expressions

It discusses the different types of operators available in C, their precedence and associativity. Mathematical functions, header files are also included.

Chapter 4: Input and Output Functions

It includes the input/output functions. Formatted functions—scanf( ), printf( ) and unformatted functions—getch( ), getche( ), getchar( ), gets( ), putch( ), putchar( ), puts( ) are explained in detail. Programming examples are also given.

Chapter 5: Control Statements

The *if* statement, *if-else* statement, *switch* statement, loops : *while* loop, *do while, for* loop, nested loops, infinite loops, jumps in loops (*break* and *continue* statement), *goto* statement and *exit( )* function are given. Programming examples are also given.

Chapter 6: Functions

It includes need for user defined functions. Defining and using functions, types of functions, passing arguments to a function : call by value, call by reference. Recursion and data storage classes are also discussed. Programming examples are also given.

Chapter 7: Preprocessors

It includes various preprocessor directives. Programming examples are also included.

Chapter 8: Arrays

It includes the meaning of an array, one dimensional and multidimensional (two or more dimensional) arrays. Different operations on arrays are given with the help of programming examples.

Chapter 9: Strings

It includes null terminated strings as array of characters and various operations on strings. User defined string functions are also given. Programming examples are also provided in the end of the chapter.

Chapter 10: Pointers

It discusses address operators, pointer data type declaration, pointer assignment, pointer initialization, pointer arithmetic, functions and pointers, arrays and pointers, pointer arrays. Command line arguments and dynamic manipulation of memory is provided here. Programming examples are also given.

Chapter 11: Structures and Unions

It includes structure variables, initialization, structure assignment, nested structure, structures and functions, structures and arrays : arrays of structures, structures containing arrays, bit fields in structures and self referential structures. An introduction to unions is also provided. Programming examples are also given.

Chapter 12: File Handling

It discusses concept of files, file opening in various modes and closing a file, reading from a file, writing onto a file. Random access is explained. Programming examples are also given.

Chapter 13: Linked Lists

It includes creation of a singly linked list, traversing a linked list, insertion into a linked list, deletion from a linked list and applications of linked lists. Programming examples are also given.

Chapter 14: Graphics in C

It discusses the concept of pixel, resolution. Various graphics functions are explained with appropriate examples. Animation is described to make the reader comfortable in graphics. Programming examples are also given.

I have put my sincere efforts and knowledge to make you understand the subject matter in the simplest and easiest form. A great care has been taken to avoid mistakes.

The author invites feedback from readers as in the past, which has encouraged me a lot in improving this book to a great extent. Please do send your comment and suggestions to the publisher or the author.

WISH YOU A GRAND SUCCESS in your examination, and a very bright future in the field of Computer Science.

—AUTHOR

Introduction to Problem Solving

Introduction

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 :

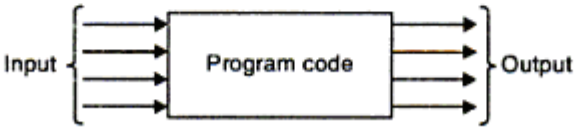


Fig. 1. *A conceptual view of a program.*

Some programs may have millions of lines of code. Building complex programs requires a disciplined approach. Most modern programs are too big for a person to write. When a group of people work together on a program and break it into separate parts that can be worked on independently, it is essential that everyone follow the same rules or programming methodologies. Over the years, there have been numerous attempts to develop methodologies that make programming more a science than art.

Planning the Computer 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.

Problem solving is one of the most significant advantages of a computer. Before writing programs, it is a good practice to understand the complete problem, analyse the various solutions and arrive at the best solution. Figure 2 illustrates the problem-solving logic.

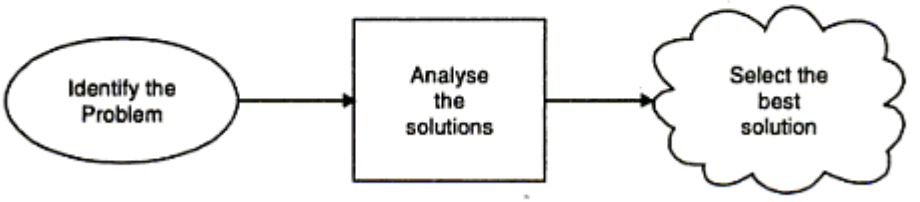


Fig. 2. *Problem-solving logic.*

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.**

Flowcharting technique is quite helpful in describing program structure and explaining it. The other useful techniques for actually designing the programs are :

(*i*) **Modular programming**

(*ii*) **Top-down design (Step wise refinement)**

(*iii*) **Structured programming.**

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.

Need of Data Structures and Algorithms

Data structure : In *computer science,* a data structure is a way of storing *data* in a computer so that it can be used efficiently.

Often a carefully chosen data structure will allow the most *efficient algorithm* to be used. The choice of the data structure often begins from the choice of an *abstract data type.* A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented by a *programming language* as *data types* and the *references* and operations they provide.

Some Examples of Data Structures

There are many types of data structures and algorithms used to create abstract data types. Here are a few of the more common examples used in computer programming.

Array

An array is a collection of homogeneous elements which is static (that is, it has fixed size). It is also known as a subscripted variable.

Stack

A stack is a container for items which can be pushed onto it, or removed from the top of it. Navigation through the stack is not normally allowed, and access is limited to one unit (or node) at a time.

Queue

A queue is similar to a stack in that items can be added or removed from it. A LIFO queue (last-in-first-out) models the same behavior as a stack, while a FIFO queue (fist-in-first-out) allows items to be added at one end, and removed from the other.

Linked List

A linked list represents a queue of items in which individual nodes can be removed, added at arbitrary points, and the whole list freely navigated. Each node is linked to the next and/or previous node.

Tree

A tree is a collection of nodes, which is dynamic in nature. One of its important types is a binary tree.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, *B-trees* are particularly well-suited for implementation of databases, while networks of machines rely on *routing tables* to function.

In the design of many types of *programs,* the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the *algorithms* to be used often become relatively obvious. Sometimes things work in the opposite direction - data structures are chosen because certain key tasks have algorithms that work best with particular data structures. In either case, the choice of appropriate data structures is crucial.

This insight has given rise to many formalised design methods and *programming languages* in which data structures, rather than algorithms, are the key organising factor. Most *languages* feature some sort of *module system,* allowing data structures to be safely reused in different applications by hiding their verified implementation details behind controlled interfaces. *Object-oriented programming languages* such as *C++* and *Java* in particular use *classes* for this purpose.

Since data structures are so crucial, many of them are included in standard libraries of modern *programming languages* and *environments,* such as *C++'s Standard Template Library containers,* the *Java* Collections Framework, and the Microsoft *.NET Framework.*

The fundamental building blocks of most data structures are *arrays, records,* and *references.* For example, the simplest linked data structure, the *linked list,* is built from records and nullable references.

Algorithm : An *algorithm* is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria :

1. Input: there are zero or more quantities which are externally supplied;
2. Output: at least one quantity is produced;
3. Definiteness: each instruction must be clear and unambiguous;
4. Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
5. Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

An algorithm can be described in several ways. One way is to give it a graphical form of notation such as flowchart. This form places each processing step in a "box" and uses arrows to indicate the next step. Different shaped boxes stand for different kinds of operations.

Algorithm Termination

A *termination proof* is a type of *mathematical proof* that plays a critical role in *formal verification* because *total correctness* of an *algorithm* depends on termination.

A simple, general method for constructing termination proofs involves associating a **measure** with each step of an algorithm. The measure is taken from the domain of a *well-founded relation,* such as from the *ordinal numbers.* If the measure "decreases" according to the relation along every possible step of the algorithm, it must terminate, because there are no *infinite descending chains* in a well-founded set.

Some types of termination analysis can automatically generate or imply the existence of a termination proof.

Example

An example of a *programming language* construct which may or may not terminate is a *Loop,* as they can be run repeatedly. Loops implemented using a *counter variable* as typically found in *data processing algorithms* will usually terminate, demonstrated by the *pseudocode* example below:

```
i := 0
loop until i = SIZE_OF_DATA
    process_data(data[i]) //process the data chunk at position i
    i := i + 1 //move to the next chunk of data to be processed
```

If the value of *SIZE_OF_DATA* is non-negative, fixed and finite, the loop will eventually terminate, assuming *process_data* terminates too.

Some loops can be shown to always terminate or never terminate, through human inspection. For example, even a non-*programmer* should see that, in theory, the following never stops (but it may halt on physical machines due to *arithmetic overflow*):

```
i := 1
loop until i = 0
    i := i + 1
```

In termination analysis one may also try to determine the termination behaviour of some program depending on some unknown input. The following example illustrates this problem.

```
i := 1
loop until i = UNKNOWN
    i := i + 1
```

Here the loop condition is defined using some value UNKNOWN, where the value of UNKNOWN is not known (*e.g.,* defined by the user's input when the program is executed). Here the termination analysis must take into account all possible values of UNKNOWN and find out that in the possible case of UNKNOWN = 0 (as in the original example) the termination cannot be shown.

There is, however, no general procedure for determining whether an expression involving looping instructions will halt, even when humans are tasked with the inspection. The theoretical reason for this is the undecidability of the Halting Problem : there cannot exist

some algorithm which determines whether any given program stops after finitely many computation steps.

In practise one fails to show termination (or non-termination) because every algorithm works with a finite set of methods being able to extract relevant information out of a given program. A method might look at how variables change with respect to some loop condition (possibly showing termination for that loop), other methods might try to transform the program's calculation to some mathematical construct and work on that, possibly getting information about the termination behaviour out of some properties of this mathematical model. But because each method is only able to "see" some specific reasons for (non)termination, even through combination of such methods one cannot cover all possible reasons for (non)termination.

Recursive functions and loops are equivalent in expression; any expression involving loops can be written using recursion, and vice versa. Thus the termination of recursive expressions are also undecidable in general. Most recursive expressions found in common usage (*i.e.*, not *pathological*) can be shown to terminate through various means, usually depending on the definition of the expression itself. As an example, the *function argument* in the recursive expression for the *factorial* function below will always decrease by 1; from the *well-ordering property* on *natural numbers*, the argument will eventually reach 1 and the recursion will terminate.

```
function factorial (argument as natural number)
    if argument = 0 or 1
        return 1
    otherwise
        return argument * factorial(argument - 1)
```

Correctness of an Algorithm

In theoretical *computer science*, **correctness** of an *algorithm* is asserted when it is said that the algorithm is correct with respect to a *specification*. Functional correctness refers to the input-output behaviour of the algorithm (*i.e.*, for each input it produces the correct output).

A distinction is made between **total correctness**, which additionally requires that the algorithm terminates, and **partial correctness**, which simply requires that *if* an answer is returned it will be correct. Since there is no general solution to the *halting problem*, a total correctness assertion may lie much deeper.

For example, if we are successively searching through *integers* 1, 2, 3, ... to see if we can find an example of some phenomenon—say an odd *perfect number*—it is quite easy to write a partially correct program (use *integer factorization* to check n as perfect or not). But to say this program is totally correct would be to assert something currently not known in *number theory*.

A proof would have to be a mathematical proof, assuming both the algorithm and specification are given formally. In particular it is not expected to be a correctness assertion for a given program implementing the algorithm on a given machine. That would involve such considerations as limitations on memory.

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, Figure 3 illustrates this concept :

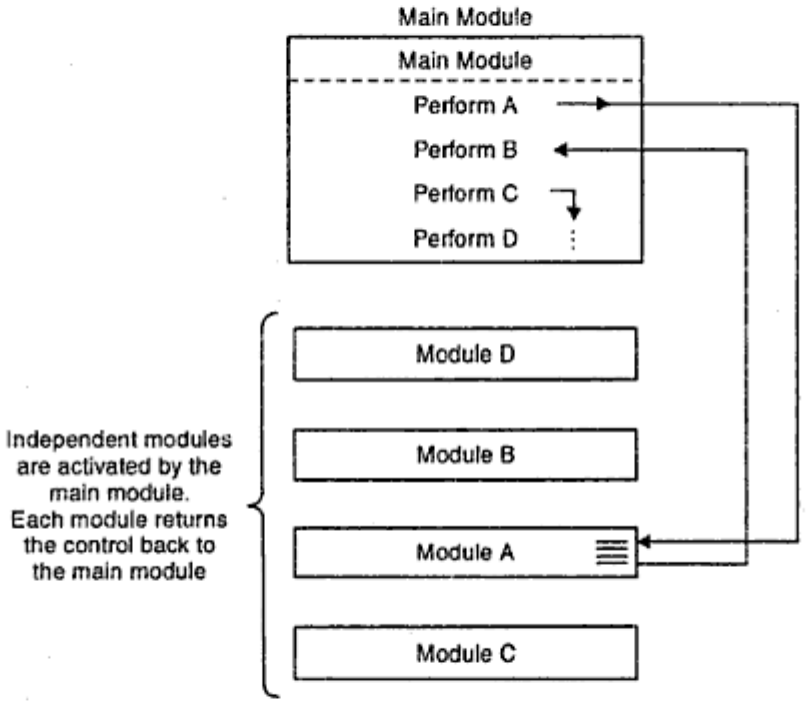


Fig. 3. *Illustration of modular approach.*

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

Characteristics of Modular Approach

The characteristics of modular approach are given below :

(*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.

Advantages of Modular Approach

The advantages of modular approach are given below :

(*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.

Top-down Design (Stepwise Refinement)

Program development includes designing, coding, testing and verification of a program in any computer language. For writing a good program, the top down design approach can be used. It is also called **systematic programming** or **hierarchical program design** or **stepwise refinement**. A complex problem is broken into smaller subproblems, further each subproblem is broken into a number of smaller subproblems and so on till the subproblems at the lowest level are easy to solve. Similarly a large program is broken into a number of subprograms and in turn each subprogram is further decomposed into subprograms ans so on. Suppose we want to solve a problem S, which can be decomposed into subproblems S1, S2 and S3 and so on. Let the program for S, S1, S2, S3 be denoted by P, P1, P2, P3 respectively. Further suppose that S2 is solved by decomposing it into subproblems S21 and S22 and program P21 and P22 are written for these. This operation of coding a subprogram in terms of lower level subprograms is known as the **process of stepwise refinement.** Figure 4 shows the hierarchical decomposition of P into its subprograms and sub-subprograms.

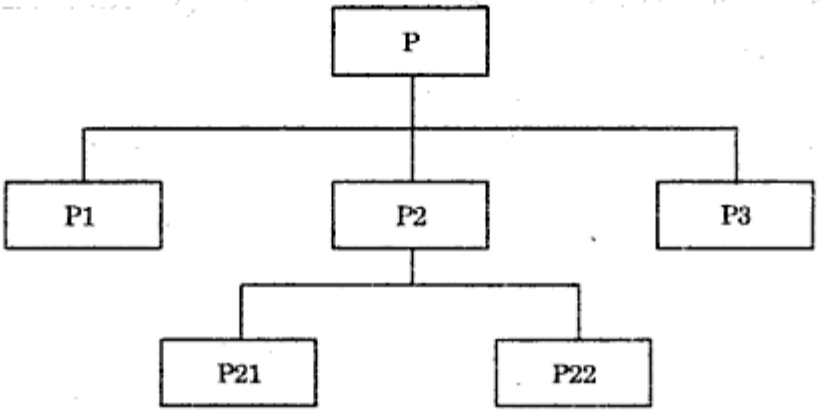


Fig. 4. *Illustration of stepwise refinement.*

The advantages of the top-down design approach are :

1. A large problem is divided into a number of smaller problems using this approach. The decomposition is continued till the subproblems at the lowest level become easy to solve. So the overall problem solving becomes easy.
2. If we use the top-down approach for a problem then top-down programming method can be used for coding modules at various stages. So, the top level modules can be coded without coding the lower level modules earlier. This approach, is better than the bottom-up approach where programming starts first at the lowest level modules.
3. It helps in top-down testing and debugging of programs.
4. The programs become user friendly (that is easy to read and understand) and easy to maintain and modify.
5. Different programmers can write the modules for different levels.

Structured Programming

Structured programming takes a top-down approach that breaks programs into modular forms. The main objectives of structured programming are :

- Efficiency
- Readability
- Clarity of programs
- Easy modification
- Reduced testing problems.

The **goto** statement should be avoided so far as possible. The three basic building blocks for writing structured programs are given below :

1. Sequence Structure
2. Loop or iteration
3. Binary Decision Structure

1. **Sequence Structure :**
   It consists of a single statement or a sequence of statements with a single entry and single exit as shown below.

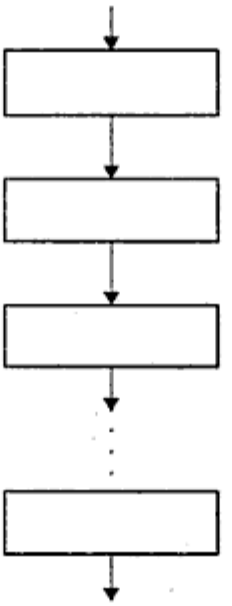


Fig. 5. *Sequence structure.*

2. **Loop or Iteration :**
   It consists of a condition (simple or compound) and a sequence structure which is executed depending on the condition based as shown below.

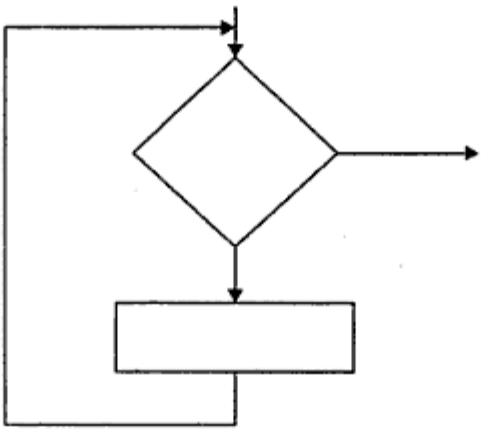


Fig. 6. *Loop or iteration.*

3. **Binary Decision Structure :**
   It consists of a condition (simple or compound) and two branches out of which

one is to be followed depending on the condition being true or false as shown below.

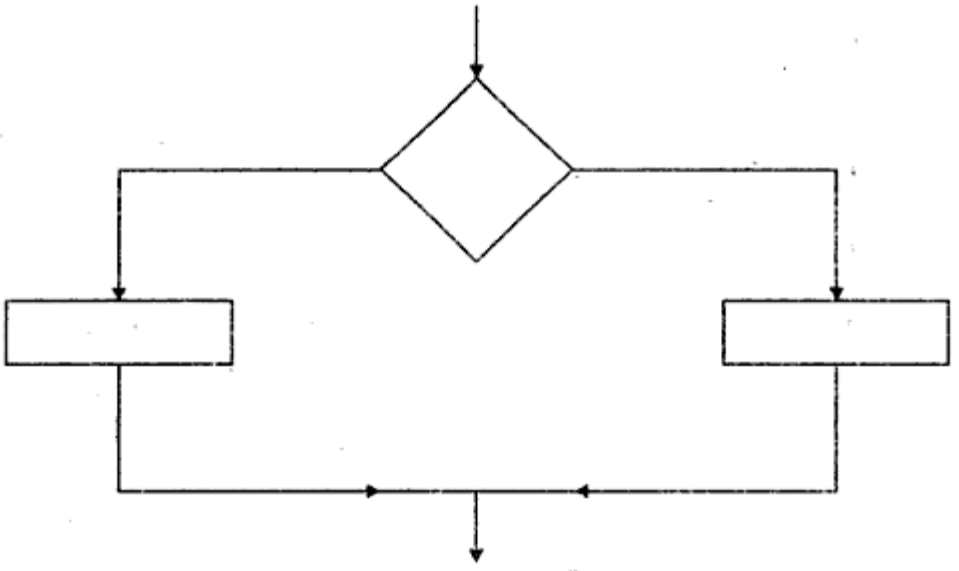


Fig 7. *Binary decision structure.*

Algorithm Design

Computers are basically used to solve complex problems in a systematic and easy manner. In order to solve a problem systematically, the solution should be written as a set of sequential steps. Each of these steps specify some simple actions that need to be performed. Thus, *an algorithm may be defined as a finite and ordered sequence of steps which when performed lead to the solution of the problem in a definite time.* Ordered sequence implies that the execution takes place in the same manner or order in which the statements are written *i.e.*, each step of the algorithm is written in such a way that the next instruction follows automatically. The ordering is provided by assigning positive integers to the steps. The words BEGIN and END normally refer to the beginning and end of the algorithm. An algorithm must possess following characteristics :

1. **Finiteness.** Finiteness implies that the algorithm must have finite number of steps. Also the time taken to execute all the steps of the algorithm should be finite and within a reasonable limit.
2. **Definiteness.** By definiteness it is implied that each step of the algorithm must specify a definite action *i.e.*, the steps should not be vague. Moreover, the steps should be such that it is possible to execute these manually in a finite length of time.
3. **Input.** The term input means supplying initial data for an algorithm. This data must be present before any operations can be performed on it. Sometimes no data is needed because initial data may be generated within the algorithm. Thus, the algorithm may have no or more inputs. Generally, the initial data, is supplied by a READ instruction or a variable can be given initial value using SET instruction.

4. **Output.** The term output refers to the results obtained when all the steps of the algorithm have been executed. An algorithm must have at least one output.
5. **Effectiveness.** Effectiveness implies that all the operations involved in an algorithm must be sufficiently basic in nature so that they can be carried out manually in finite interval of time.

Expressing Algorithms

The procedure for expressing algorithm is quite simple. The language used to write algorithms is similar to our day-to-day life language. In addition, some special symbols are also used which are described below :

(*i*) **Assignment symbol (←).** The assignment symbol (←) is used to assign values to various variables. For example, let A be any variable and B be another variable or constant or an expression. Then the statement

$$A \leftarrow B$$

is called assignment statement. The statement implies that A is assigned the value stored in B. If A contains any previous value then that value is destroyed and the new value is assigned.

(*ii*) **Relational symbols.** The commonly used relational symbols for algorithms are :

Symbol	*Meaning*	*Example*
<	Less than	A < B
<=	Less than or equal to	A <= B
=	Equal to	A = B
≠	Not equal to	A ≠ B
>	Greater than	A > B
>=	Greater than or equal to	A >= B

(*iii*) **Brackets ({}).** The pair of braces is used to write comments for the purpose of documentation.

For example,

(*i*) BEGIN {Start of the algorithm}

(*ii*) Set N ← N + 1 {Increase the value of N by 1}

(*iii*) END {End of the algorithm}.

Basic Control Structures

The basic control structures needed for writing good and efficient algorithms are :

(*i*) *Selection*

(*ii*) *Branching*

(*iii*) *Looping.*

(*i*) **Selection.** The selection structure is used when we have to perform a given set of instructions if the given condition is TRUE and an alternative set of

instructions if the condition is FALSE. The basic statement available for selection is IF-THEN-ELSE.

The syntax is :

```
IF (condition is true) THEN
{
        s1
        s2
        .
        .
        .
        sn
}
ELSE
{
        f1
        f2
        .
        .
        .
        fn
}
```

For example, consider the following algorithm which finds greater among 2 numbers.

```
            BEGIN
STEP 1      Read NUM1, NUM2
STEP 2      IF NUM1 > NUM2 THEN
            Write (NUM1, "is greater")
            ELSE
            Write (NUM2, "is greater")
            END
```

(*ii*) **Branching.** The branching statement is required when we want to transfer the control of execution from one part or step of the algorithm to another part or step. The statement available for branching is GOTO and its syntax is :

GOTO *n*

where *n* is a positive integer and specifies the step number where the control of execution is to be transferred.

(*iii*) **Looping.** The looping structure is used when a statement or a set of statements is to be executed a number of times. The following two loop control structures are commonly used in algorithms :

(*a*) WHILE-DO

(*b*) REPEAT-UNTIL

(*a*) **WHILE-DO :** The syntax is :

```
STEP 1      WHILE (Condition) DO
STEP 2      S1
STEP 3      S2
            .
            .
            .
STEP N+1    SN
STEP N+2    END-WHILE {End of While-Do loop}
```

This control loop structure implies that as long as the condition remains true, all the steps listed between WHILE-DO and END-WHILE are executed again and again. As soon as the condition becomes false, the execution of the loop stops and control is transferred to next statement following END-WHILE.

For example, consider the following algorithm

```
            BEGIN
STEP 1      Set N ← 1
STEP 2      WHILE (N <= 10) DO
STEP 3      Write N
STEP 4      Set N ← N + 1
STEP 5      END-WHILE
            END
```

This algorithm initially sets the value of N to 1. The while statement then checks if the value of N <= 10. If the condition is true it executes steps 3 and 4. When the value of N exceeds 10 the condition becomes false and the control goes to the statement following END-WHILE which is END statement marking the END of algorithm.

(*b*) **REPEAT-UNTIL :** This is similar to WHILE-DO except the fact that the loop is executed till the condition remains false or condition becomes true. The syntax is :

```
STEP 1      REPEAT
STEP 2      S1
STEP 3      S2
            .
            .
            .
STEP N + 1  SN
STEP N + 2  UNTIL (Condition)
```

This control loop structure implies that as long as the condition remains false, all the steps listed between REPEAT and UNTIL are executed again and again. As soon as the condition becomes true, the execution of the loop stops and control is transferred to next statement following UNTIL (condition). *For example,* consider the following algorithm :

```
                BEGIN
STEP 1          Set N ← 1
STEP 2          REPEAT
STEP 3          Write N
STEP 4          Set N ← N + 1 {Increment N by 1}
STEP 5          UNTIL (N > 10)
                END
```

Initially, the value of N is set to 1. The loop executes till the value of N exceeds 10. After this, the control goes to the next statement following UNTIL (N > 10).

Advantages of Algorithms

The main advantages of algorithms are given below :

(*i*) It is simple to understand step by step solution of the problem.

(*ii*) It is easy to debug *i.e.,* errors can be easily pointed out.

(*iii*) It is independent of programming languages.

(*iv*) It is compatible to computers in the sense that each step of an algorithm can be easily coded into its equivalent in high-level language.

Development of Algorithms for Simple Problems

ALGORITHM : Exchanging Values of Two Variables

Given two variables A and B. We have to exchange these variables. TEMP is used for exchanging the variables.

```
1. INPUT A, B
2. TEMP ← A
       A ← B
       B ← TEMP
3. Write "Exchanged values are ", A, B
4. End.
```

ALGORITHM : Biggest of Three Numbers

Given the three numbers A, B, C. We have to find the biggest of these.

```
1. INPUT A, B, C
2. IF (A > B) THEN
   Begin
        IF (A > C) THEN
             Write "Biggest number is ", A
        ELSE
             Write "Biggest number is ", C
   End
   ELSE
```

```
Begin
        IF (B > C) THEN
                Write "Biggest number is ", B
        ELSE
                Write "Biggest number is ", C
End
```

3. END.

ALGORITHM : Area of a Triangle

Given three sides A, B, C of a triangle. We have to find the area (if possible). S denotes the semi-perimeter.

1. INPUT A, B, C
2. **IF** (((A+B) > C) AND ((B+C) > A) AND ((C+A) > B)) **THEN**

```
Begin
        S ← (A + B + C)/2
        Area ← √(S × (S – A) × (S – B) × (S – C))
        Write "Area of triangle is ", Area, " sq. units"
End
ELSE
        Write "Triangle is not possible"
```

3. END.

ALGORITHM : Roots of a Quadratic Equation ($Ax^2 + Bx + C = 0$)

1. INPUT A, B, C
2. **IF** A = 0 **THEN**

```
Begin
        IF B = 0 THEN
        Begin
                Write "Equation is degenerate"
                goto step 5
        End
        ELSE
        Begin
                Write "Linear equation has single root"
                x1 = –C/B
                Write "Root = ", x1
                goto step 5
        End
End
```

3. D = B × B – 4.0 × A × C
4. **IF** D > 0 **THEN**

```
Begin
    Write "Real and distinct roots"
    x1 = (–B + √D ) / (2.0 × A)
    x2 = (–B – √D ) / (2.0 × A)
    Write "First root = ", x1
    Write "Second root = ", x2
End
ELSE
Begin
    IF (D = 0) THEN
    Begin
        Write "Real and equal roots"
        x1 = –B / (2.0 × A)
        x2 = x1
        Write "First root = ", x1
        Write "Second root = ", x2
    End
    ELSE
    Begin
        Write "Imaginary roots"
        x1 = –B / (2.0 × A)
        x2 = x1
        y1 = √–D / (2.0 × A)
        y2 = –y1
        Write "First root "
        Write "Real part ", x1, " Img. part ", y1
        Write "Second root "
        Write "Real part ", x2, " Img. part ", y2
    End
End
```

5. END.

ALGORITHM : Sum of Series 1 + 5 + 9 + 13 + ... 20 Terms

Given the series 1 + 5 + 9 + 13 + ... We have to find the sum of 20 terms. NUM is a temporary variable for storing a term. SUM denotes the sum of terms. Variable COUNT is used as loop computer.

1. SUM ← 0
2. NUM ← 1
3. Repeat for COUNT = 1, 2, ..., 20

Begin

SUM ← SUM + NUM

NUM ← NUM + 4

End

4. Write "Sum of 20 terms is ", SUM
5. End.

ALGORITHM : Summation of a Set of Numbers

Given N numbers. We have to find the sum of these numbers. NUM is a temporary variable for storing a number. SUM denotes the sum of numbers. Variable COUNT is used as loop counter.

1. Read N
2. SUM ← 0
3. Repeat for COUNT = 1, 2, ..., N

   Begin

   Read NUM

   SUM ← SUM + NUM

   End
4. Write "Sum of inputted numbers is ", SUM
5. End.

ALGORITHM : Reversing Digits of an Integer

Given an integer NUM. We have to reverse digits of this integer. Variables Q and R denote quotient and remainder respectively.

1. Read NUM
2. Write "Reversed number is "
3. WHILE (NUM ≠ 0) DO

   Begin

   Q ← Integral part of (NUM/10)

   R ← NUM – (Q × 10)

   Write R

   NUM ← Q

   End
4. End.

ALGORITHM : Traversal in an Array

Given an array A of N elements. We have to traverse the array elements one by one. I denotes the array index. Let **CHANGE** denote the desired operation to be performed.

1. I ← 1
2. While (I ≤ N) DO

   Begin

```
        Apply CHANGE on A[I]
        I ← I + 1
    End
3.  END
```

OR

```
1.  Repeat for I = 1, 2, ....., N
        Apply CHANGE on A[I]
2.  END.
```

ALGORITHM : Reverse Order of Elements of an Array

Given an array A of N elements. This algorithm reverses the order of array elements. I, MID denote array indices. TEMP is used for swapping of elements.

```
1.  MID ← Integral part of (N/2)
2.  I ← 1
3.  Repeat for I = 1, 2, ....., MID
    Begin
        TEMP ← A[I]
        A[I] ← A[N – I + 1]
        A[N – I + 1] ← TEMP
    End
4.  END.
```

ALGORITHM : Concatenation of Two Arrays

Given two arrays A and B of size M and N, respectively. This algorithm concatenates the arrays in C having size M + N. I denotes array index. Let us assume that A is housed first in C and then B.

```
1.  Repeat for I = 1, 2, ....., M
        C[I] ← A[I]
2.  Repeat for I = 1, 2, ....., N
        C[M+I] ← B[I]
3.  END.
```

ALGORITHM : Find Largest of N Numbers in an Array

Given an array A having N elements. This algorithm finds the LARGEST of the elements. I denotes array index.

```
1.  LARGEST ← A[1]
2.  Repeat for I = 2, 3, ....., N
    Begin
        IF (A[I] > LARGEST) THEN
            LARGEST ← A[I]
    End
3.  Write "Largest number is ", LARGEST
4.  END.
```

ALGORITHM : Sum and Average of N Numbers in an Array

Given an array A of N elements. This algorithm finds the sum and average of all the numbers. I denotes array index.

1. SUM ← 0
2. Repeat for I = 1, 2, ..., N
   SUM ← SUM + A[I]
3. AVG ← SUM/N
4. Write SUM, AVG
5. END.

ALGORITHM : Linear Search

Given an array A of N elements. This algorithm searches for an element DATA in the array. I denotes the array index. This algorithm gives the first location where DATA is found.

1. I ← 1
2. While (I ⩽ N) DO upto step 3
3. **IF** (A[I] = DATA) **THEN**
   Begin
   Write "Successful search "
   Write DATA, " found at position ", I
   goto step 5
   End
   ELSE
   Begin
   I ← I + 1
   End
4. Write "Unsuccessful search"
5. END.

ALGORITHM : Binary Search (Always Applicable on Sorted Data)

Given an array A of N elements in ascending order. This algorithm searches for an element DATA. LOW, HIGH, MID denote the lowest, highest and middle position of a search interval respectively.

1. LOW ← 1
   HIGH ← N
2. While (LOW ⩽ HIGH) DO upto step 4
3. MID ← Integral part of ((LOW + HIGH)/2)
4. **IF** (DATA = A[MID]) **THEN**
   Begin
   Write "Successful search"
   Write DATA, " found at position ", MID
   goto step 6

```
        End
        ELSE
        Begin
            IF (DATA > A[MID]) THEN
                LOW ← MID + 1
            ELSE
                HIGH ← MID – 1
        End
5.  Write "Unsuccessful search"
6.  END.
```

ALGORITHM : Organize Numbers in Ascending Order

Given an array A of N elements. This algorithm arranges the elements in ascending order. I and J denote array indices. Variable TEMP is used for swapping.

```
1.  Repeat for I = 1, 2, ....., N–1
    Begin
        Repeat for J = I + 1, I + 2, ....., N
        Begin
            IF (A[J] < A[I]) THEN
            Begin
                TEMP ← A[I]
                A[I] ← A[J]
                A[J] ← TEMP
            End
        End
    End
2.  END.
```

ALGORITHM : Selection sort *(For Ascending Order)*

Given an array A of N elements. This algorithm arranges the elements in ascending order. PASS denotes pass counter. MINDEX denotes the position of the least element during a pass. I denotes array index. Variable TEMP is used for swapping.

```
1.  Repeat for PASS = 1, 2, ....., N–1 upto step 4
2.  MINDEX ← PASS
3.  Repeat for I = PASS + 1, PASS + 2, ..... N
    Begin
        IF (A[I] < A[MINDEX]) THEN
            MINDEX ← I
    End
4.  IF (PASS ≠ MINDEX) THEN
    Begin
        TEMP ← A[PASS]
```

```
        A[PASS] ← A[MINDEX]
        A[MINDEX] ← TEMP
    End
5.  END
```

ALGORITHM : Bubble sort *(For Ascending Order)*

Given an array A of N elements. This algorithm arranges the elements in ascending order. PASS denotes pass counter. LAST denotes the position of the last unsorted element during a particular pass. EXCHS denotes the number of exchanges made during any pass. I denotes array index. Variable TEMP is used for swapping.

```
1.  LAST ← N
2.  Repeat for PASS = 1, 2, ....., N–1 upto step 5
3.  EXCHS ← 0
4.  Repeat for I = 1, 2, ....., LAST–1
    Begin
        IF (A[I] > A[I+1]) THEN
        Begin
            EXCHS ← EXCHS + 1
            TEMP ← A[I]
            A[I] ← A[I+1]
            A[I+1] ← TEMP
        End
    End
5.  IF EXCHS = 0 THEN
        goto step 6
    ELSE
        LAST ← LAST - 1
6.  END
```

ALGORITHM : Insertion sort *(For Ascending Order)*

Given an array A of N elements. This algorithm arranges the elements in ascending order. CURRENT denotes the element to be placed at it's proper position. I, J, POS denote array indices.

```
1.  Repeat for I = 2, 3, ....., N upto step 5
2.  CURRENT ← A[I]
3.  POS ← 1
4.  Repeat While ((POS < I) AND (A[POS] < CURRENT))
        POS ← POS + 1
5.  IF (POS ≠ I) THEN
    Begin
        Repeat for J = I–1, I–2, ....., POS
        Begin
            A[J+1] ← A[J]
```

```
        End
        A[POS] ← CURRENT
    End
6.  END
```

ALGORITHM : Merging of two arrays

Given two arrays A and B of size M and N respectively in ascending order of elements. This algorithm merges the two arrays in C of size M + N, in ascending order. I, J, K denote array indices.

```
1.  I ← 1
    J ← 1
    K ← 1
2.  Repeat While ((I ⩽ M) AND (J ⩽ N))
    Begin
        IF (A[I] ⩽ B[J]) THEN
        Begin
            C[K] ← A[I]
            I ← I + 1
        End
        ELSE
        Begin
            C[K] ← B[J]
            J ← J + 1
        End
        K ← K + 1
    End
3.  IF (I > M) THEN
    Begin
        Repeat While (J ≤ N)
        Begin
            C[K] ← B[J]
            J ← J + 1
            K ← K + 1
        End
    End
    ELSE
    Begin
        Repeat While (I ≤ M)
        Begin
            C[K] ← A[I]
            I ← I + 1
```

```
                K ← K + 1
            End
        End
4.  END
```

ALGORITHM : Insertion in an array at a specific position

Given an array A of M elements having size N (M < N). This algorithm inserts an element DATA at the position POS (POS ⩽ M+1). I denotes array index. After insertion there will be M + 1 elements.

```
1.  Repeat for I = M, M–1, ....., POS
        A[I+1] ← A[I]
2.  A[POS] ← DATA
3.  M ← M + 1
4.  END
```

ALGORITHM : Insertion in a sorted array

Given an array A of M elements in ascending order of size N (M < N). This algorithm inserts an element DATA so that after insertion, the order of elements is preserved (*i.e.*, remains sorted). I and POS denote array indices. After insertion there will be M+1 elements.

```
1.  IF (DATA ⩾ A[M]) THEN
    Begin
        A[M+1] ← DATA
        goto step 6
    End
2.  I ← 1
3.  Repeat While (A[I] ⩽ DATA)
        I ← I + 1
4.  Repeat for POS = M, M–1, ....., I
        A[POS+1] ← A[POS]
5.  A[I] ← DATA
6.  M ← M + 1
7.  END
```

ALGORITHM : Matrix Multiplication

Given two matrices A and B of orders m × n and p × q respectively. This algorithm multiplies the two matrices (if possible) and stores the result in matrix C (of order m×q). I, J, K denote array indices.

```
1.  IF n ≠ p THEN
    Begin
        Write "Matrix multiplication not possible"
        goto step 3
    End
```

```
2.  Repeat for I = 1, 2, ....., m
    Begin
        Repeat for J = 1, 2, ....., q
        Begin
            C[I,J] ← 0
            Repeat for K = 1, 2, ....., n
            C[I,J] ← C[I,J] + (A[I,K] × B[K,J])
        End
    End
3.  END.
```

Flowcharting

The technique of drawing *flowcharts* is known as flowcharting. A flowchart is a pictorial representation of the sequence of operations necessary to solve a problem with a computer. The first formal flowchart is attributed to *John Von Neumann* in 1945. The flowcharts are read from left to right and top to bottom. Program flowcharts show the sequence of instructions in a program or a subroutine. The symbols used in constructing a flowchart are simple and easy to learn. These are very important planning and working tools in programming. The purposes of the flowcharts are given below :

1. Provide better communication

These are an excellent means of communication. The programmers, teachers, students, computer operators and users can quickly and clearly get ideas and descriptions of algorithms.

2. Provide an overview

A clear overview of the complete problem and its algorithm is provided by the flowchart. Main elements and their relationships can be easily seen without leaving important details.

3. Help in algorithm design

The program flow can be shown easily with the help of a flowchart. A flowchart can be easily drawn in comparison to writing a program and testing it. Different algorithms (for the same problem) can be easily experimented with flowcharts.

4. Check the program logic

All the major portions of a program are shown by the flowchart, precisely. So, the accuracy in logic flow is maintained.

5. Help in coding

A program can be easily coded in a programming language with the help of a flowchart. All the steps are coded without leaving any part so that no error lies in the code.

6. Modification becomes easy

A flowchart helps in modification of an already existing program without disrupting the program flow.

7. Better documentation provided

A flowchart gives a permanent storage of program logic pictorially. It documents all the steps carried out in an algorithm. A comprehensive, carefully drawn flowchart is always an indispensable part for the program's documentation.

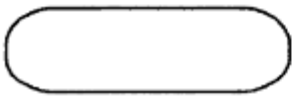
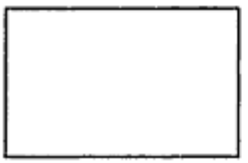
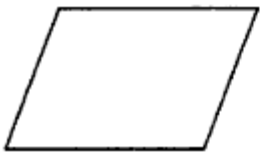
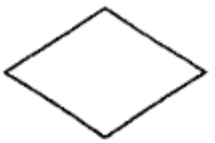
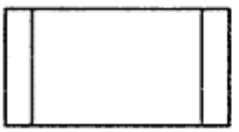
Flowchart Symbols

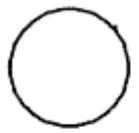
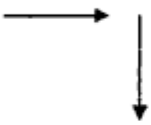
Flowcharts have only a few symbols of different sizes and shapes for showing necessary operations. Each symbol has specific meaning and function in a flowchart. These symbols have been standardized by the *American National Standards Institute (ANSI)*. The basic rules that a user must keep in mind while using the symbols are :

1. Use the symbols for their specific purposes.
2. Be consistent in the use of symbols.
3. Be clear in drawing the flowchart and the entries in the symbols.
4. Use the annotation symbol when beginning a procedure.
5. Enter and exit the symbols in the same way.

The flowchart symbols alongwith their purposes are given in Table 1.

Table 1. Flowchart Symbols

	Terminal	Indicates the beginning and end of a program.
	Process	For calculation or assigning of a value to a variable.
	Input/Output (I/O)	Any statement that causes data to be input to a program (INPUT, READ) or output from the program, such as printing on the display screen or printer.
	Decision	Program decisions. Allows alternate courses of action based on a condition. A decision indicates a question that can be answered *yes* or *no* (or *true* or *false*).
	Predefined Process	A group of statements that together accomplish one task. Used extensively when programs are broken into modules.

	Connector	Can be used to eliminate lengthy flowlines. Its use indicates that one symbol is connected to another.
	Flowlines and Arrowheads	Used to connect symbols and indicate the sequence of operations. The flow is assumed to go from top to bottom and from left to right. Arrowheads are only required when the flow violates the standard direction.
	Annotation	Can be used to give explanatory comments.

Program Control Structures

A control structure, or logic structure, is a structure that controls the logical sequence in which computer program instructions are executed. In structured program design, three control structures are used to form the logic of a program : sequence, selection and iteration (or loop). These are also used for drawing flowcharts.

Let us consider the three control structures :

(1) In the ***sequence control structure*, one program statement follows another in logical order.** There are no decisions to make, no choices between "yes" or "no". The boxes logically follow one another in sequential order. For example, figure 8 illustrates a sequence of N statements :

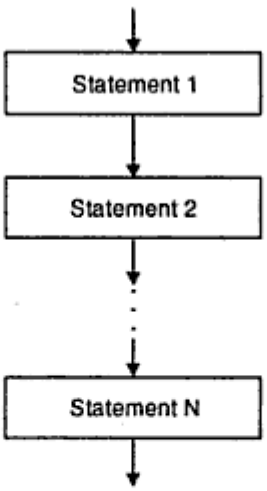


Fig. 8. *Sequence control structure.*

(2) *The selection control structure—also known as an IF-THEN-ELSE structure—represents a choice. It offers two paths to follow when a decision must be made by a program.* An example of a selection structure is as follows :

IF a student's marks in a subject is ⩾ 60 **THEN**
Student has secured first division

ELSE

Student has not secured first division

Figure 9 illustrates the selection control structure.

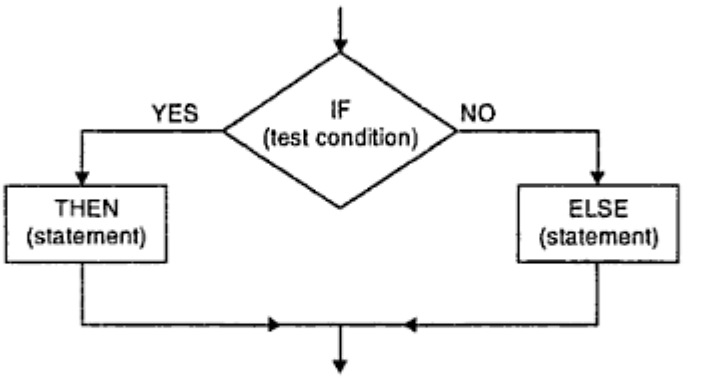


Fig. 9. *Selection control structure (IF-THEN-ELSE).*

A variation on the usual selection control structure is the *case control structure.* This offers more than a single yes-or-no decision. The case structure allows several alternatives, or "cases", to be presented. "IF Case 1 occurs, THEN do thus-and-so. IF Case 2 occurs, THEN follow an alternative course...." And so on. The case control structure saves the programmer the trouble of having to indicate a lot of separate IF-THEN-ELSE conditions. Figure 10 illustrates this :

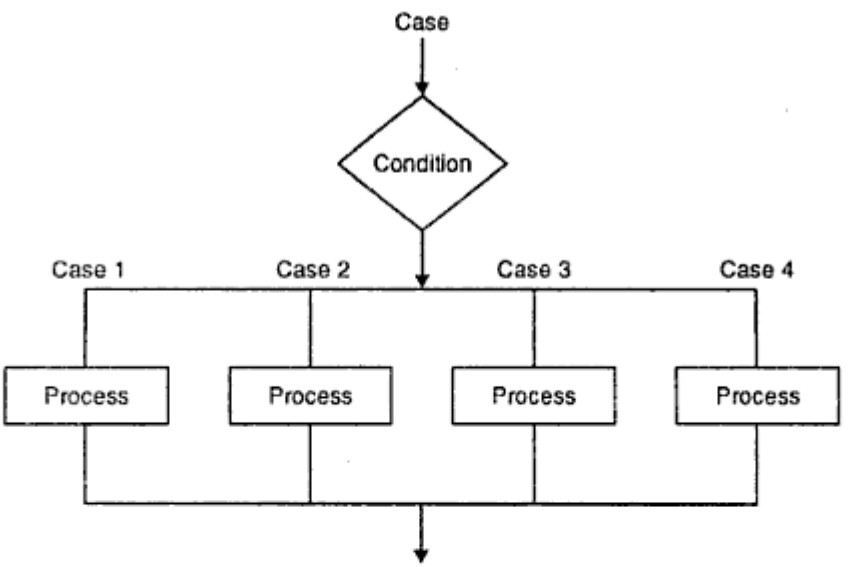


Fig. 10. *Variation on selection : the case control structure.*

(3) In the ***iteration,*** or ***loop, control structure,*** **a process may be repeated as long as a certain condition remains true.** There are two types of iteration structures—*REPEAT-UNTIL* and *WHILE-DO.* Of these, REPEAT-UNTIL is more often encountered.

An example of a REPEAT-UNTIL structure is as follows :

REPEAT read in student records UNTIL there are no more student records.

An example of a WHILE-DO structure is as follows :

WHILE read in student records DO—that is, as long as—there continue to be student records.

The difference between the two iteration structures is : If several statements are to be repeated, we must decide when to *stop* repeating them. **WHILE-DO** structure can be used to stop them at the *beginning* of the loop. Or we can decide to stop them at the *end* of the loop using **REPEAT-UNTIL** structure. REPEAT-UNTIL iteration means that the loop statements will be executed at least once, because the condition is tested in the end of loop.

One thing that all three control structures have in common is *one entry* and *one exit*. The control structure is entered at a single point and exited at another single point. This helps simplify the logic so that it is easier for others following in a programmer's footsteps to make sense of the program.

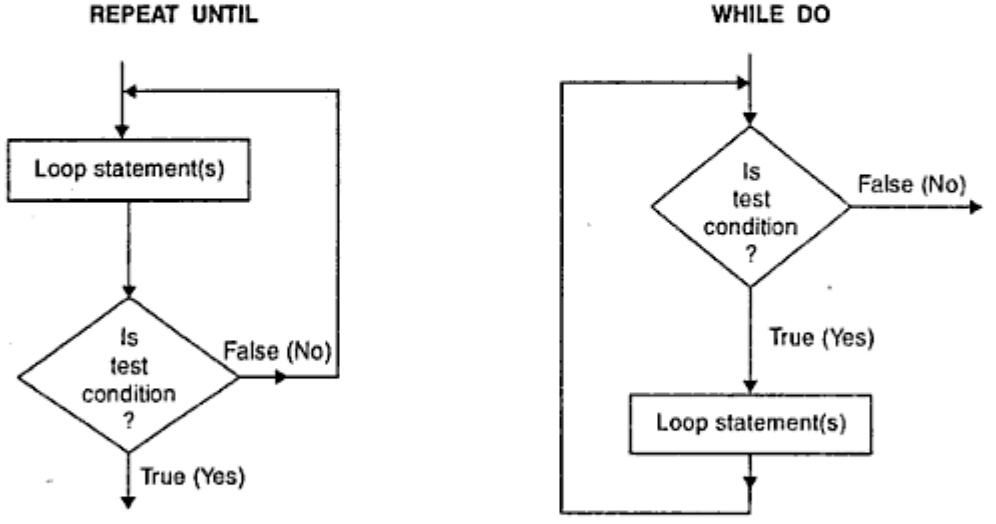


Fig. 11. *Iteration control structures (Loops).*

Types of Flowcharts

The systems designer and programmer use the following types of flowcharts in developing algorithms :

1. System flowcharts
2. Modular program flowcharts
3. Detail program flowcharts or application flowcharts

1. System flowchart

It plays a vital role in the system analysis. A system is a group of interrelated components tied together according to a plan to achieve a predefined objective. The elements and characteristics of a system are graphically shown and its structure and relationship are also

represented by flowchart symbols. The system analysts use the system flowcharts for analysing or designing various systems. The different stages of a system are :

(*i*) Problem recognition

(*ii*) Feasibility

(*iii*) System analysis

(*iv*) System design

(*v*) Implementation

(*vi*) Evaluation

All the above stages use the system flowchart for convenience. Any alternative solution for the existing system or the entirely new system can be systematically represented. The working system is well documented by a precisely drawn system flowchart.

2. Modular program flowchart

A system flowchart indicates the hardware, identifies the various files and represents the general data flow. A modular program flowchart on the other hand defines the logical steps for the input, output and processing of the information of a specific program. In structured or modular programs, the independent modules or units are written for different procedures. This module is useful in performing the specified operation in other programs too. The exact operation in detail is not performed but only the relationship and order in which processes are to be performed are included.

It is also called as block diagram. Its main advantage lies in the fact that the programmer can concentrate more on flow of logic and temporarily computer level details are ignored. Alternate algorithms without much time consumption or effort can also be tried using it. These help a lot in communicating the main logic of the program.

3. Detail program flowchart

These are the most comprehensive and elemental charts in developing the programs. The symbols represented by it are quite useful for coding the program in any computer language. Each computer language has its own syntax, so there may be some difference in performing the operations to be followed for coding a program. A detail program flowchart represents each minute operation in its proper sequence, reduced to its simplest parts.

Rules for Drawing Flowcharts

The following rules and guidelines are recommended by ANSI for flowcharting :

1. First consider the main logic, then incorporate the details.
2. Maintain a consistent level of detail for a flowchart.
3. Do not include all details in a flowchart.
4. Use meaningful descriptions in the flowchart symbols. These should be easy to understand.
5. Be consistent in using variables and names in the flowchart.
6. The flow of the flowchart should be from top to bottom and from left to right.
7. For a complex flowchart, use connectors to reduce the number of flow lines. The crossing of lines should be avoided as far as possible.

8. If a flowchart is not drawn on a single page, it is recommended to break it at an input or output point and properly labelled connectors should be used for linking the portions of the flowchart on separate pages.
9. Avoid duplication so far as possible.

Levels of Flowcharts

There are two levels of flowcharts :

(*i*) Macro flowchart

(*ii*) Micro flowchart

(*i*) **Macro flowchart.** It shows the main segment of a program and shows lesser details.

(*ii*) **Micro flowchart.** It shows more details of any part of the flowchart.

> ***Note :*** *A flowchart is independent of all computer languages.*

Examples of Flowcharts

The following figure depicts the flowchart for converting celsius temperature to fahrenheit using the formula $°F = \frac{9}{5} \times °C + 32.0$.

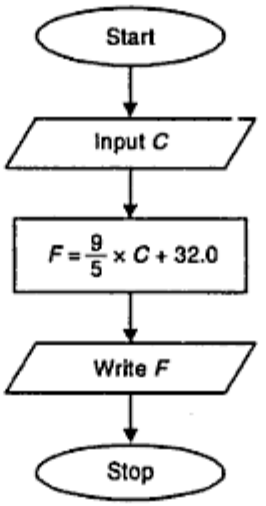


Fig. 12.

The following figure depicts the flowchart for finding the area and circumference of a circle, whose radius *r* is given.

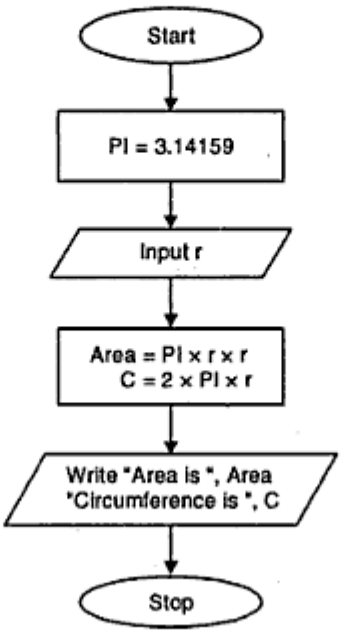


Fig. 13.

The following figure depicts the flowchart for swapping (interchanging) two numbers using a temporary variable.

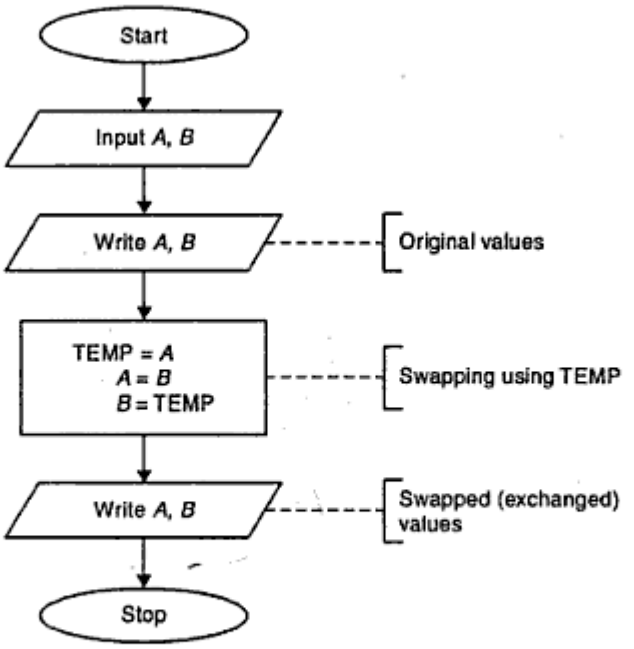


Fig. 14.

The following figure depicts the flowchart for swapping two numbers without using a temporary variable.

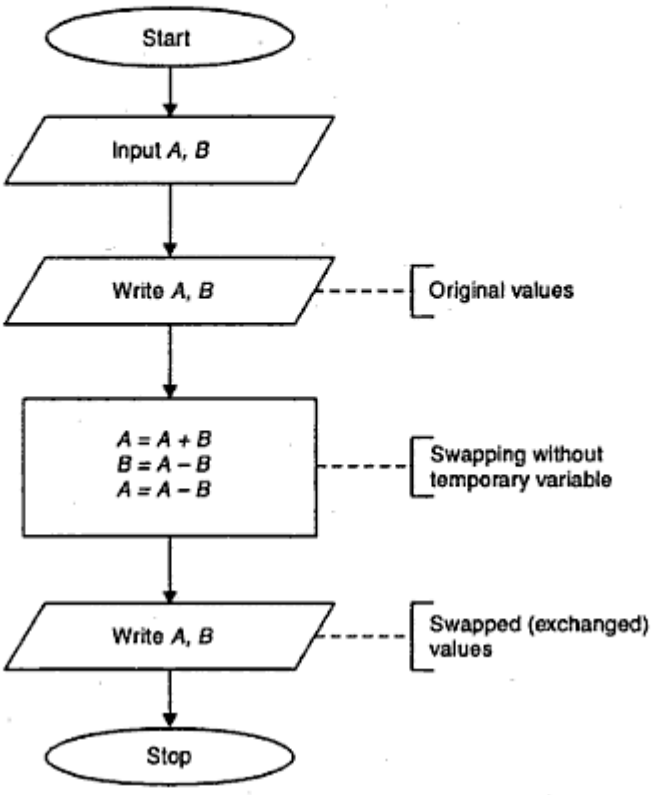


Fig. 15.

The following figure depicts the flowchart for checking a year for leap year. (If a year is divisible by 4 and not divisible by 100, or, if the year is divisible by 400, it is a leap year). Here **mod** is used for finding remainder.

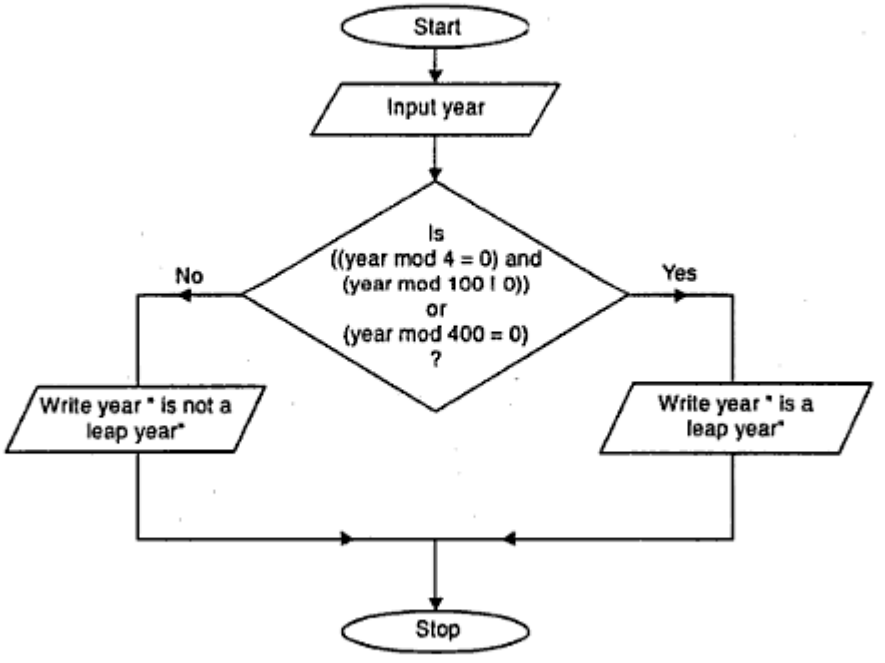


Fig. 16.

The following figure depicts the flowchart to find out the biggest of three numbers.

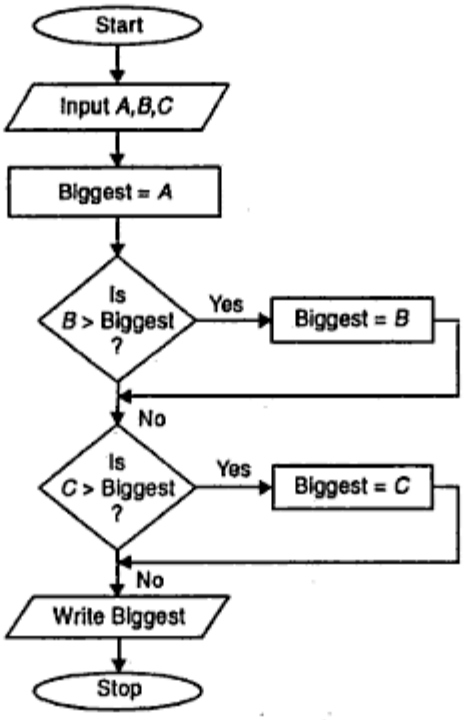


Fig. 17.

The flowchart for finding the biggest of three numbers can also be drawn as given below :

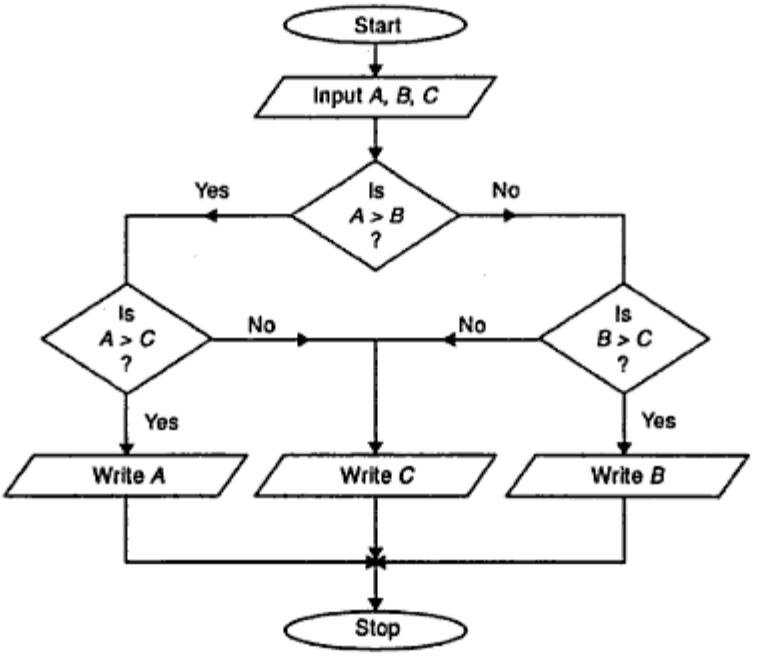


Fig. 18.

The following figure depicts the flowchart for solving a given quadratic equation.

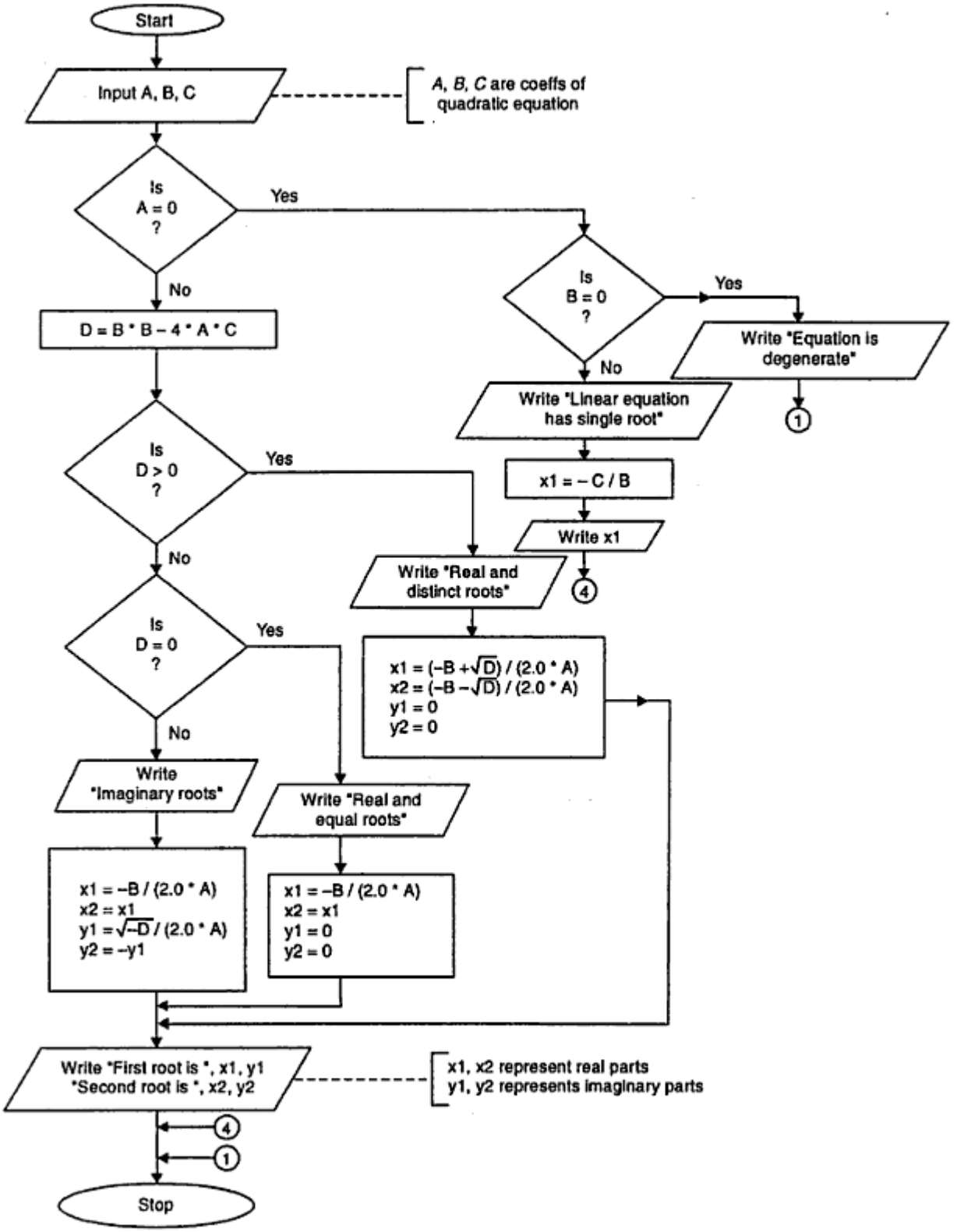


Fig. 19.

The following figure illustrates the flowchart for finding the area of a triangle and its type (that is equilateral, isosceles or scalene).

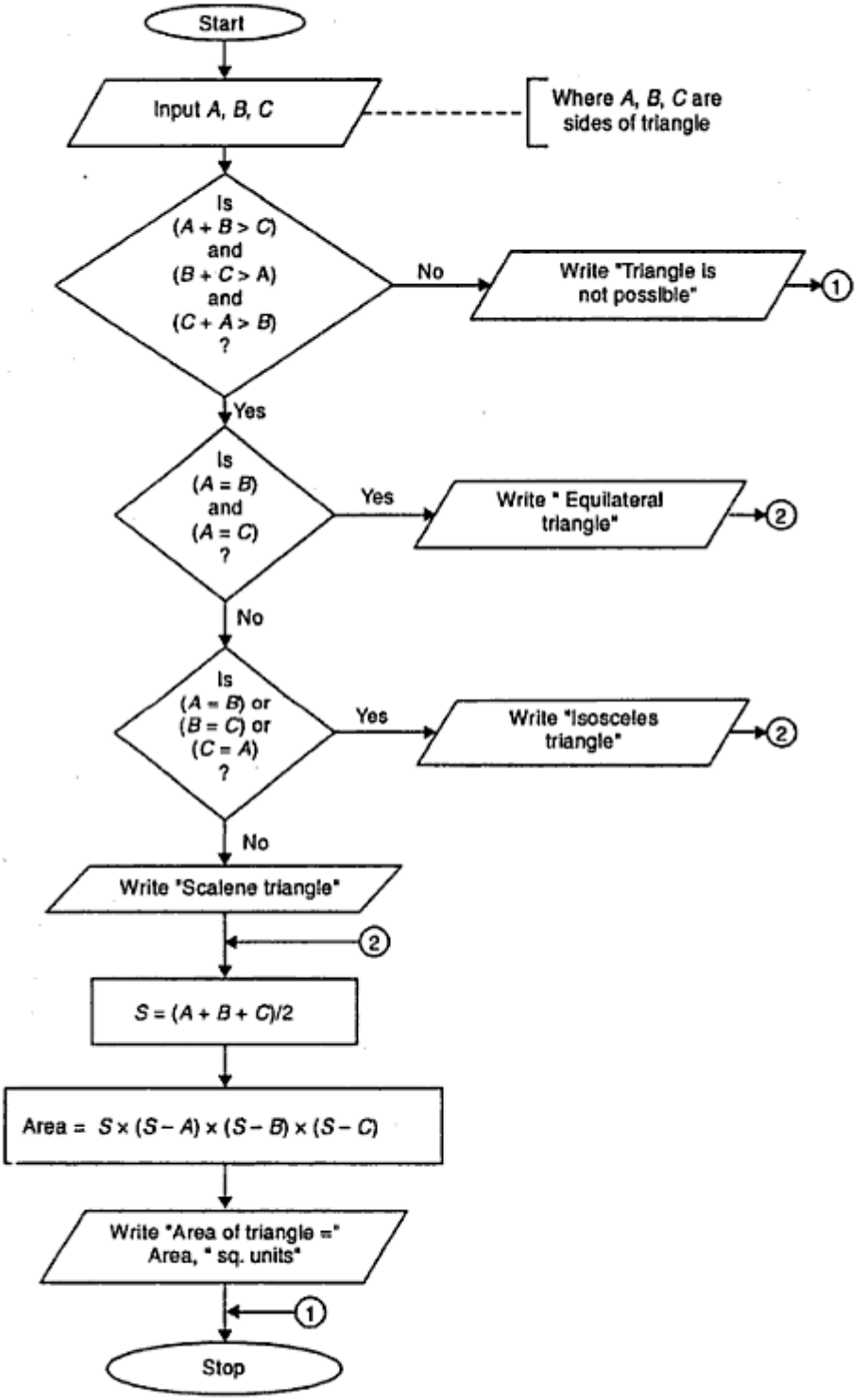


Fig. 20.

The following figure depicts the flowchart for simulating a simple calculator (that is performing +, −, ×, /)

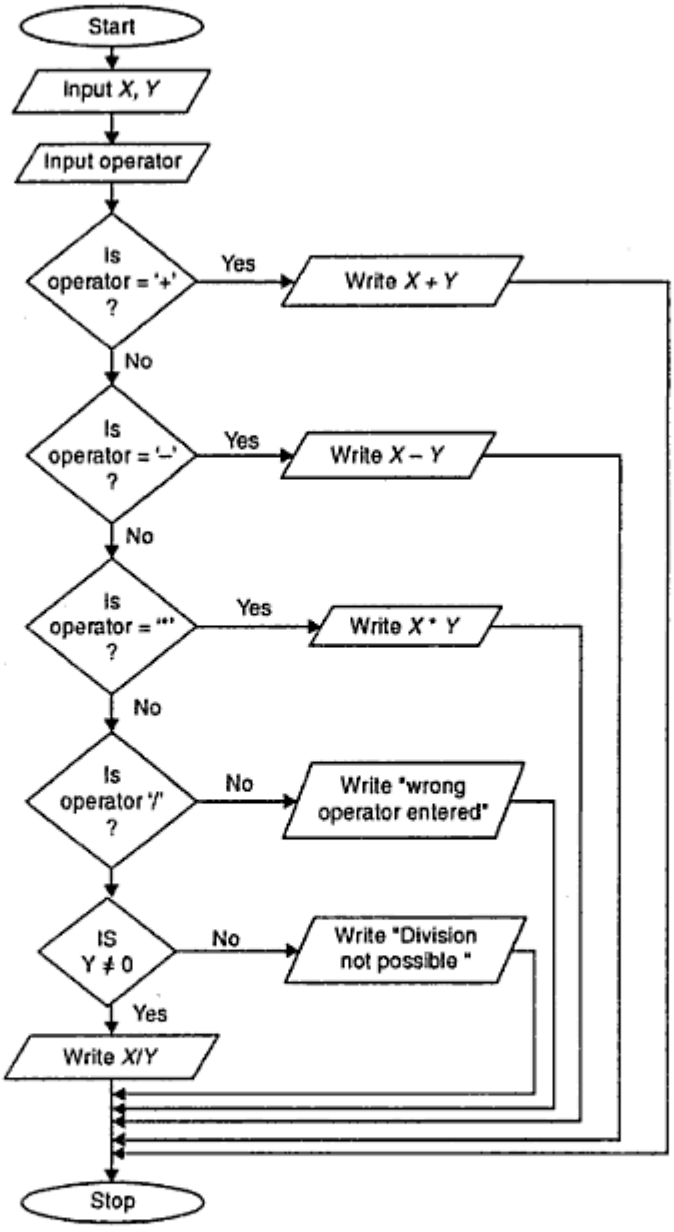


Fig. 21.

The following figure depicts the flowchart for printing the multiplication table of an integer.

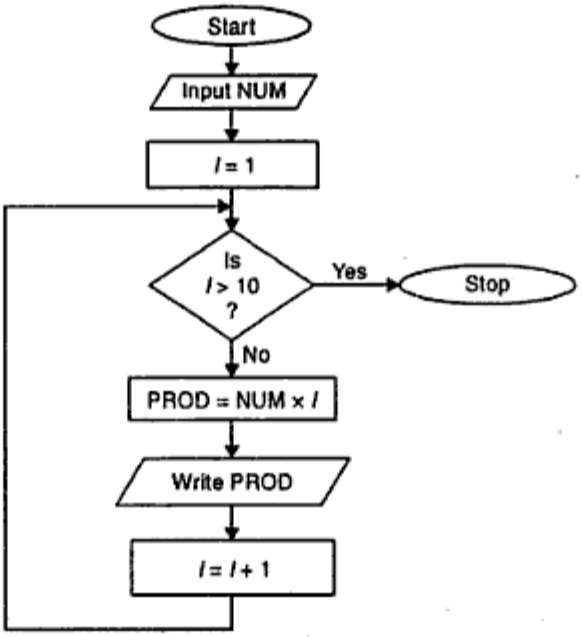


Fig. 22.

The following figure depicts the flowchart for finding the sum of first N natural Numbers.

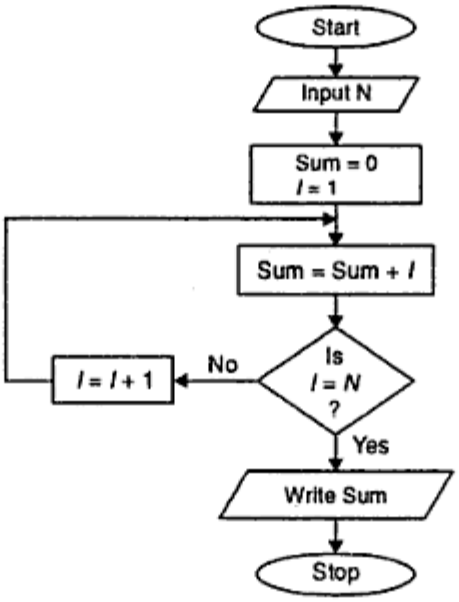


Fig. 23.

The following figure depicts the flowchart for finding the factorial of a given number.

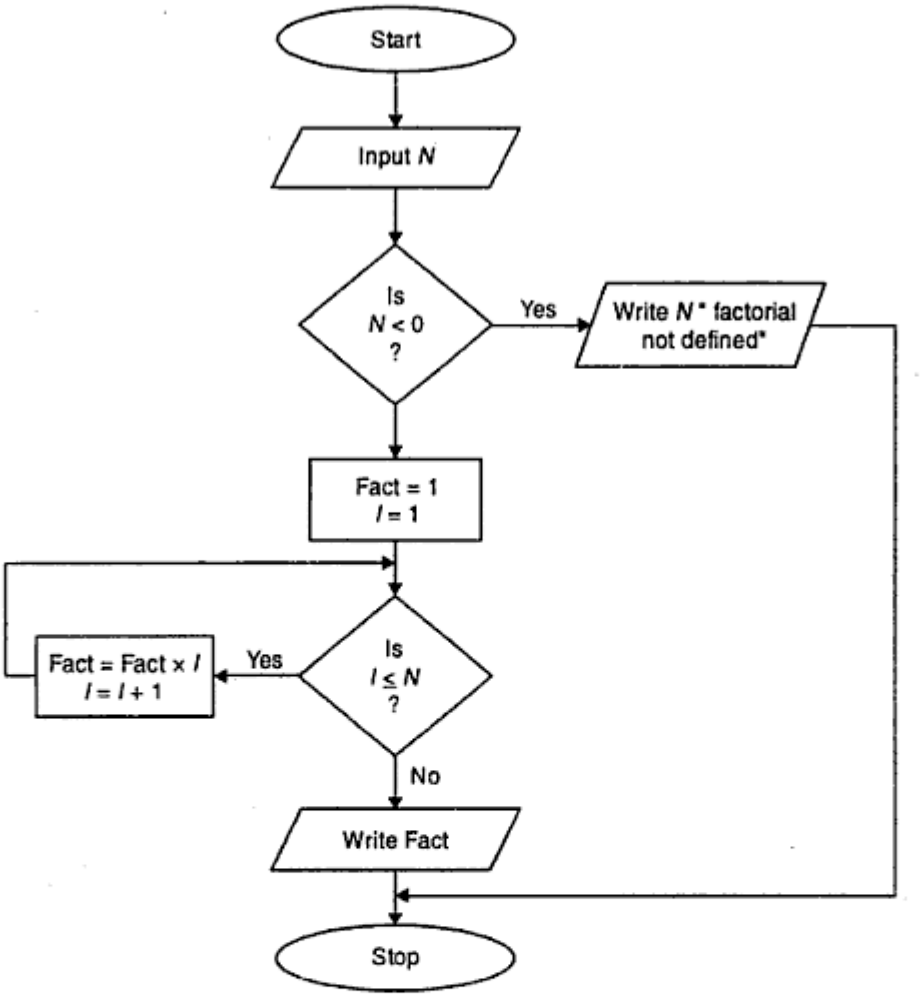


Fig. 24.

The following figure depicts the flowchart for generating first n (n ≥ 2) fibonacci terms (Fibonacci terms are 0, 1, 1, 2, 3, 5, 8, 13, 21 .....).

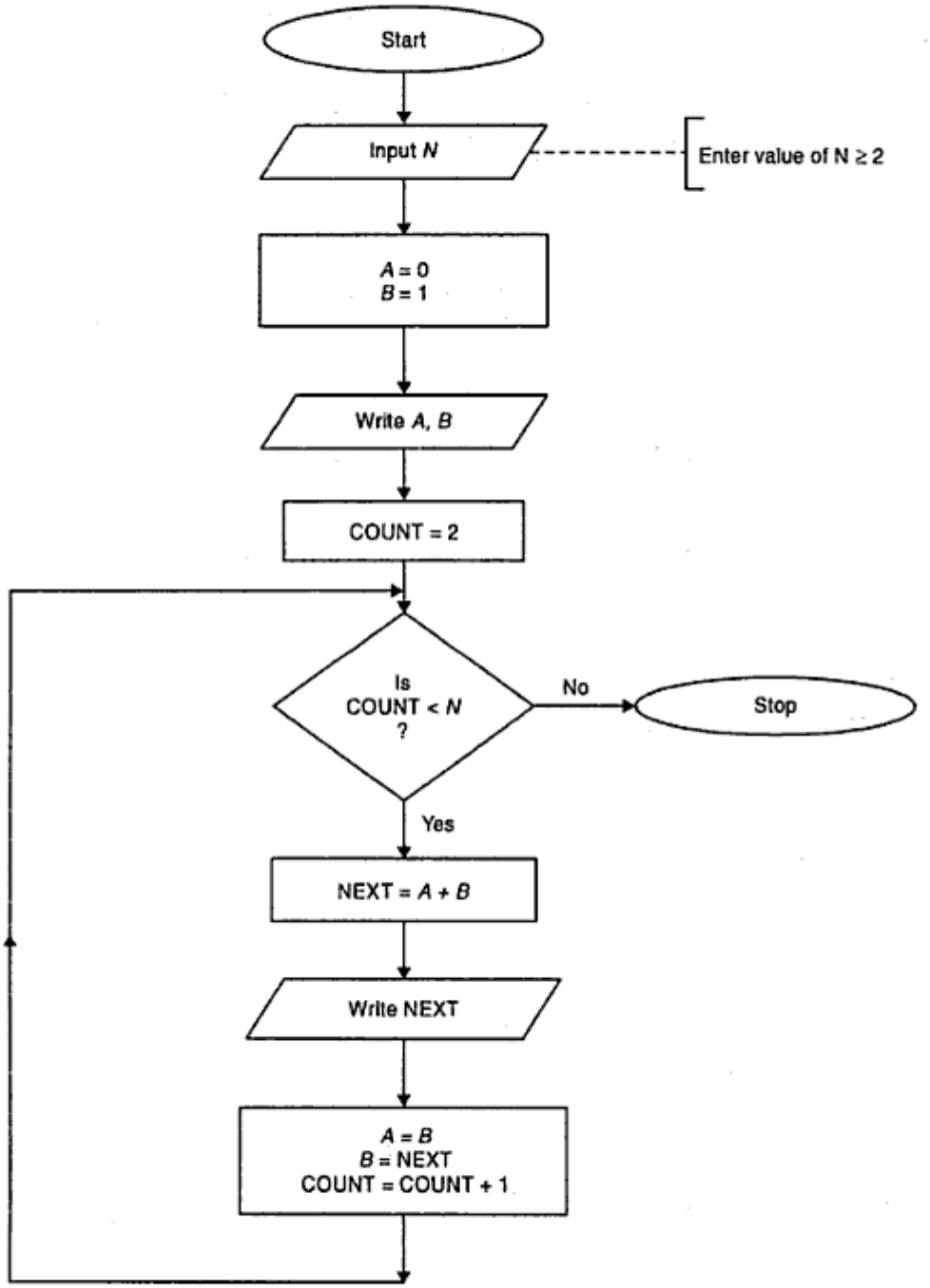


Fig. 25.

The following figure depicts the flowchart for checking an integer for prime number (A number p > 1 whose only divisors are 1 and itself is a prime number).

> ***Note** : A number is prime if it is not divisible by any integer greater than 1 and less than or equal to the integral part of its square root.*

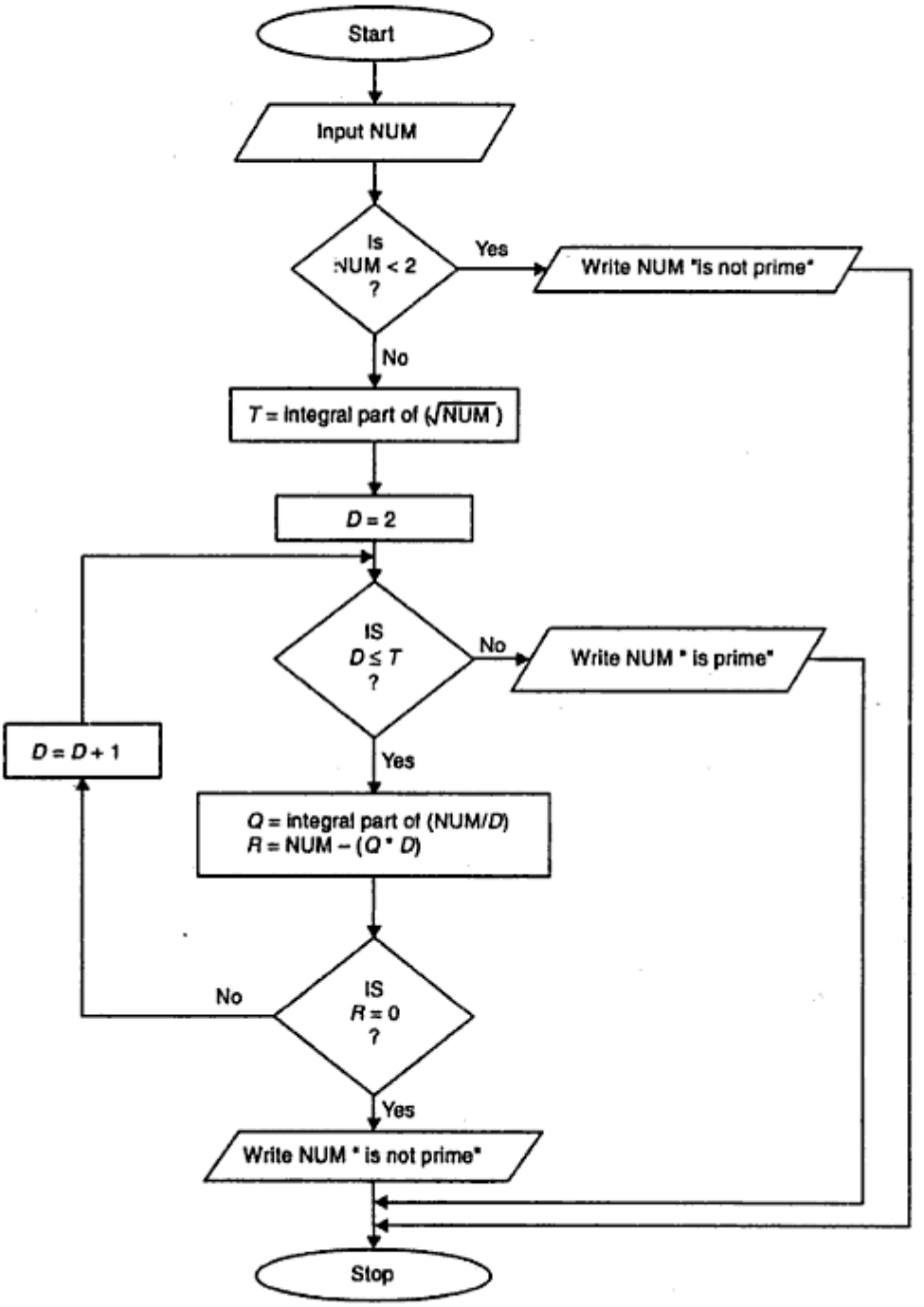


Fig. 26.

The following figure depicts the flowchart for finding the biggest of given set of numbers in an array.

> ***Note :*** *The subscript is started from 1, it may be the same or start from 0 in some of the computer languages when you code the program.*

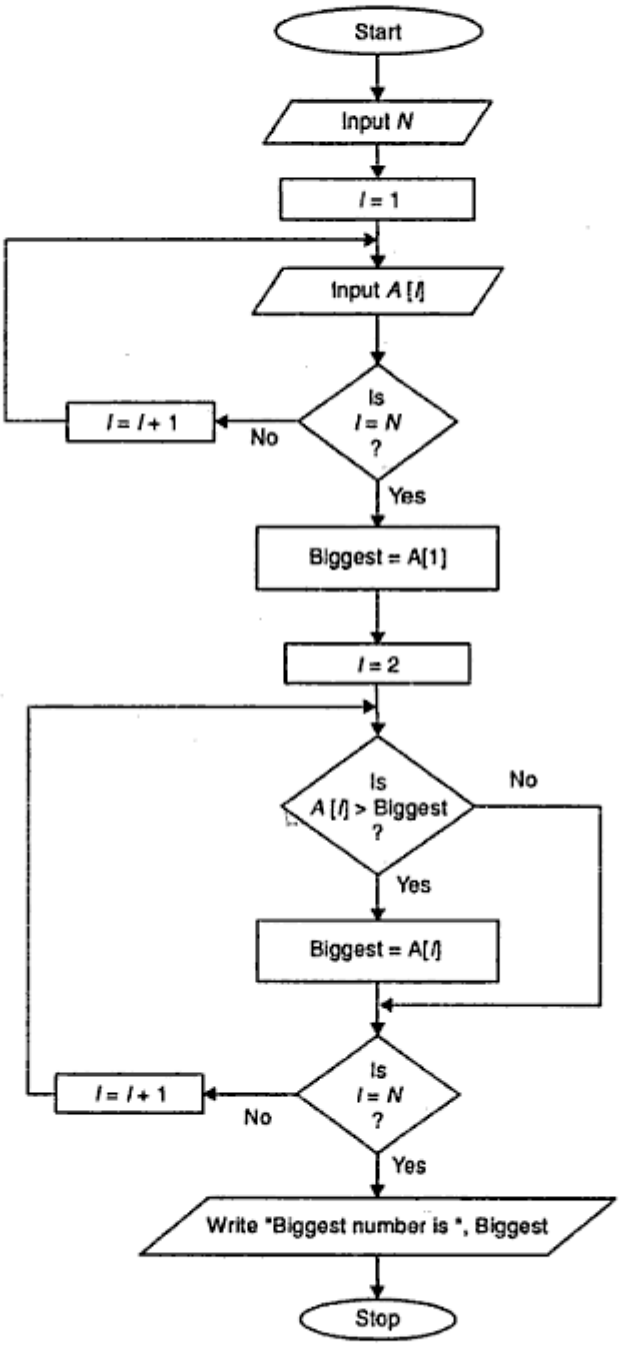


Fig. 27.

The following figure depicts the flowchart for searching an element from an array using linear search method (only first occurrence will be searched).

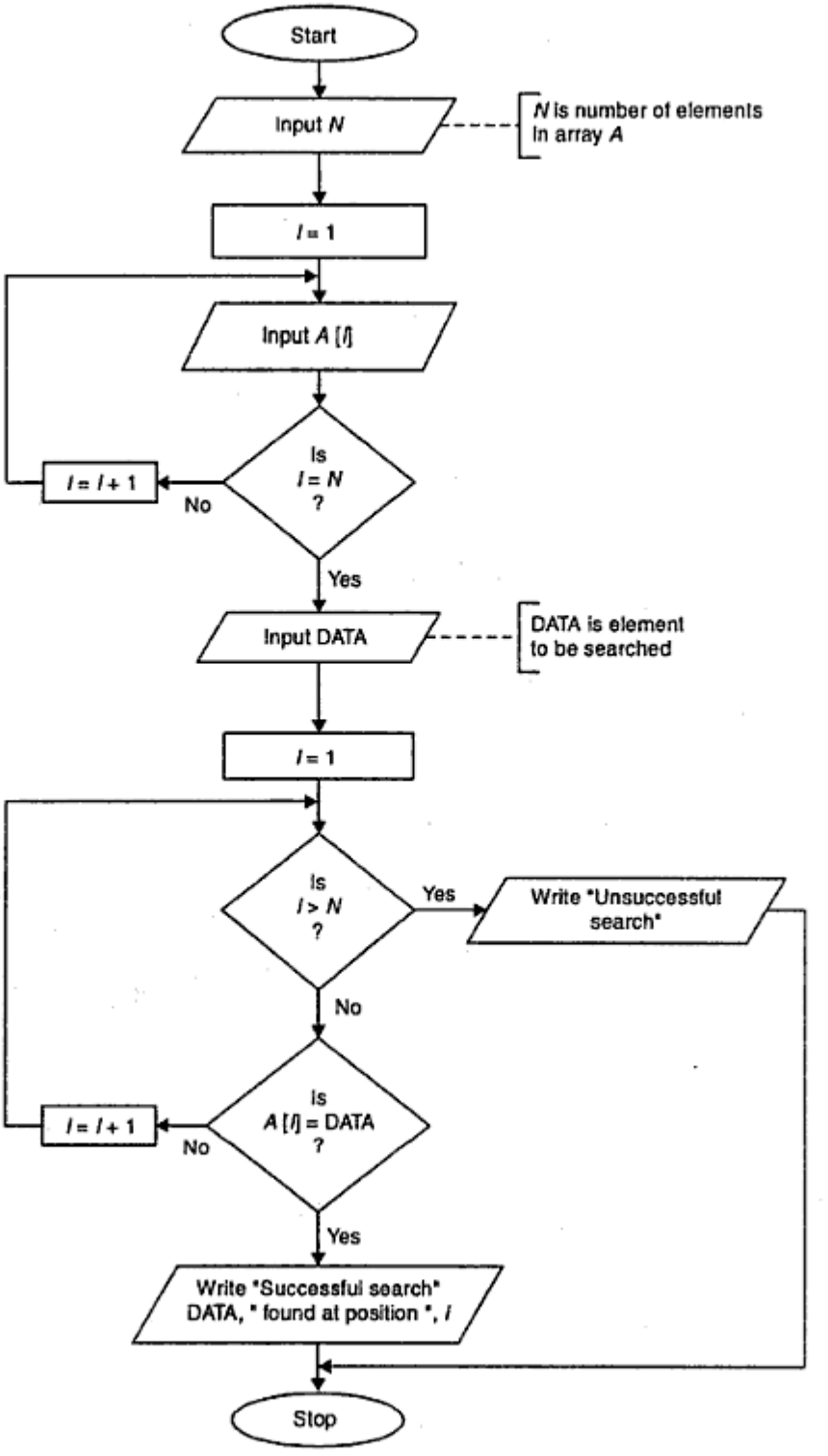


Fig. 28.

The following figure depicts the flowchart of organizing numbers in ascending order.

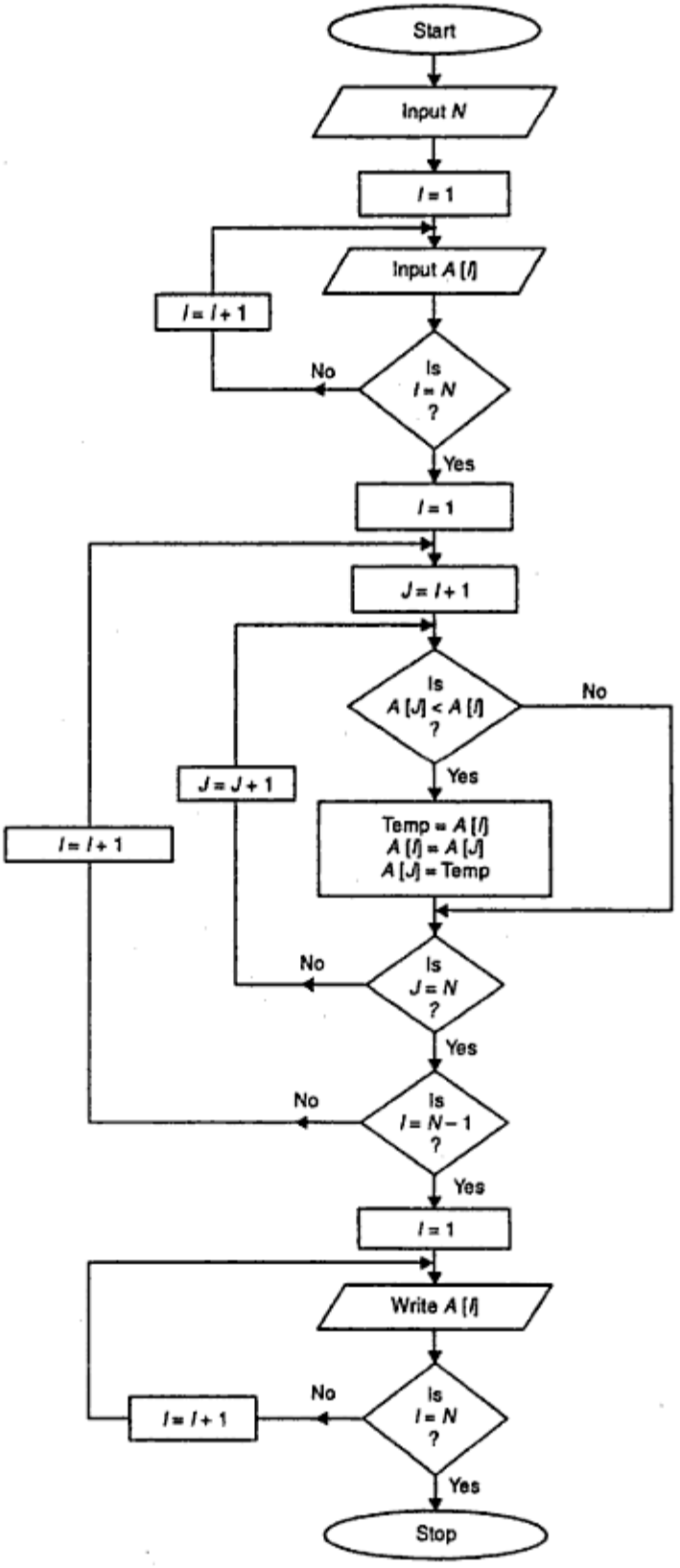
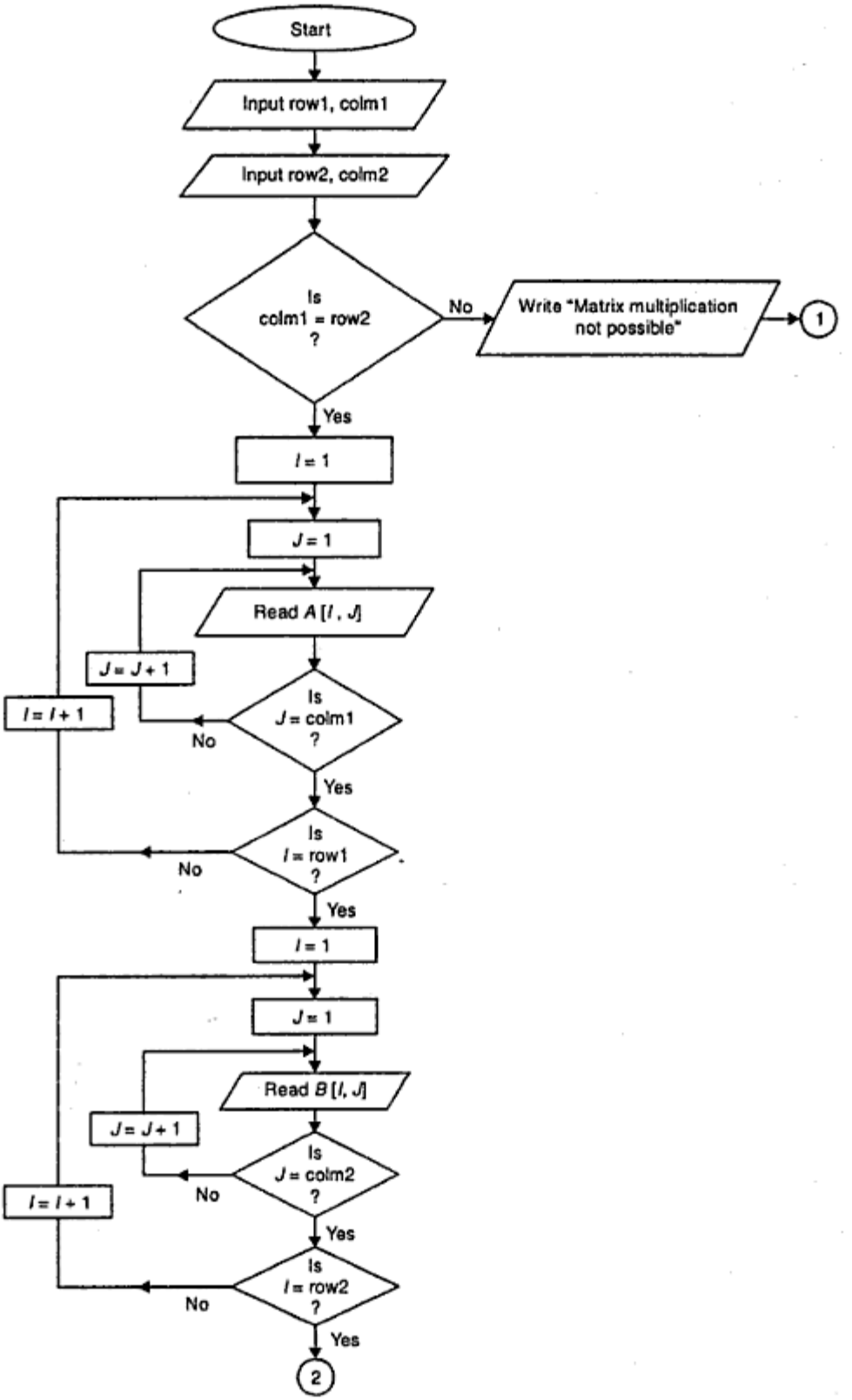


Fig. 29.

The following figure depicts the flowchart for multiplication of two matrices.



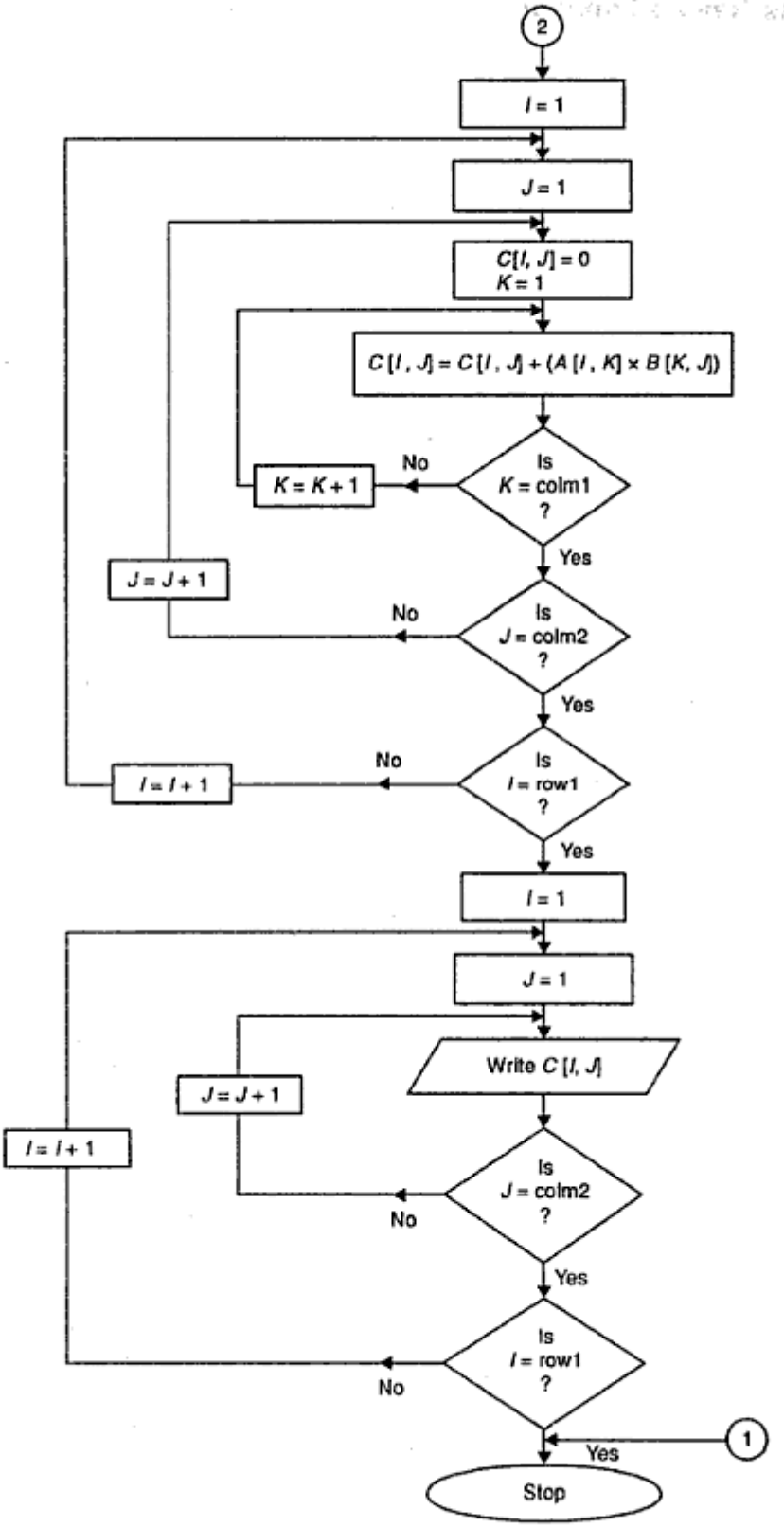


Fig. 30.

Flowcharts Having Functions

Problems can be broken up into sections so complex tasks are completed through a sequence of simpler steps. One way of doing this is by using functions. Functions can be written once and then used over and over again as the need arises. For example, the following flowchart converts fahrenheit temperature to centigrade using the formula $°C = \frac{5}{9} \times (°F - 32)$.

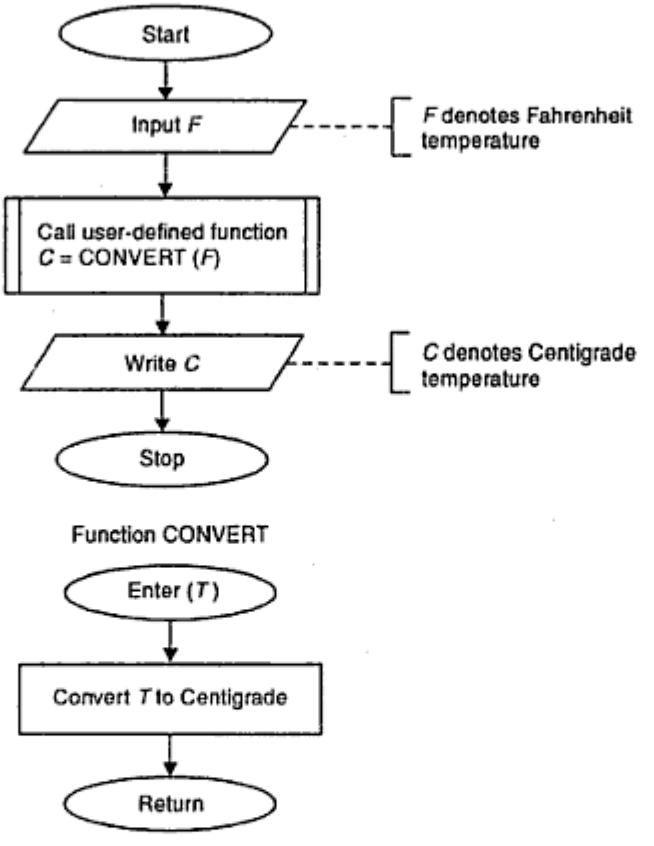


Fig. 31. *Flowchart for converting fahrenheit temperature to centigrade temperature.*

Limitations of Flowcharting

Flowcharts have some limitations also. These are given below :

Time consuming

These take a lot of time and are laborious to draw with proper symbols and spacing, especially for large complex programs.

Difficult to modify

Any change or modification in the logic of the program generally requires a completely new flowchart for it. Redrawing a flowchart is tedious and many organizations either do not modify it or draw the flowchart using a computer program.

No standard available

There are no standards provided all over the world for the details to be included in drawing a flowchart. So different people draw flowchart with different views.

Pseudocode

Pseudocode is a method of designing a program using normal human-language (an English language) statements to describe the logic and processing flow.

OR

Pseudocode is a program analysis tool that is used for planning the logic of a program. "Pseudo" means imitation or false and "code" refers to the instructions coded in a programming language. So it is an imitation of the actual computer instructions. These pseudo instructions are phrases written in one of the ordinary natural language (*e.g.*, English, German, French etc.). It is also known as **Program Design Language** (PDL) as the main emphasis is on the design of the program. The structure of the pseudocode resembles with the computer instructions and no symbols are used like flowcharts in it.

The following basic logic structures are sufficient for writing any computer program and the pseudocode is composed of all these :

1. Sequence
2. Selection (IF.....THEN or IF.....THEN.....ELSE)
3. Loop or Iteration (DO.....WHILE or REPEAT.....UNTIL)

1. Sequence

Sequence logic is used when all the instructions are followed one by one in the order in which these are written. The pseudocode instructions are written in the sequence in which these are to be executed in the program. In pseudocode the flow of logic is from top to bottom. The pseudocode for the sequence structure is depicted below :

```
. . . . . .
. . . . . .
. . . . . .
Procedure–1
Procedure–2
Procedure–3
. . . . . .
. . . . . .
. . . . . .
```

(Pseudocode for sequence structure)

2. Selection

Selection logic or decision logic, is used for making decisions using simple or compound conditions. It is used for selecting one proper path out of the two or more alternative paths at our disposal in the program logic. It is depicted in one of the two following forms :

IF.....THEN

IF.....THEN.....ELSE

Pseudocode for IF.....THEN.....ELSE

```
.....
.....
```

```
IF CONDITION THEN
    Procedure–1
    Procedure–2
    Procedure–3
ELSE
    Procedure–1
    Procedure–3
    Procedure–2
ENDIF
.....
.....
```

Here, if the CONDITION is true then the following procedures are performed : Procedure–1, Precedure–2, Procedure–3 else the precedures Procedure–1, Procedure–3, Procedure–2 are performed.

Pseudocode for IF.....THEN

```
.....
.....
IF CONDITION THEN
    Procedure–1
    Procedure–2
    Procedure–3
ENDIF
.....
.....
```

Here, if the CONDITION is true then Procedure–1, Procedure–2, Procedure–3 are performed otherwise the instrcutions written after ENDIF are performed without performing any of these procedures. The instructions will also be followed after ENDIF when the condition is true but after performing the three precedures. ENDIF indicates the end of the decision structure.

3. Loop or Iteration

Iteration logic is used when the instructions are to be executed many times depending on some condition (s). The two structures used for it are :

DO.....WHILE

REPEAT.....UNTIL

Pseudocode for DO.....WHILE

```
.....
.....
DO WHILE CONDITION
    Procedure–1
```

```
        Procedure–2
        Procedure–3
            .
            .
            .
        Procedure–n
ENDDO
.....
.....
```

Here, the looping will continue as long as the CONDITION remains true and the control comes out of the loop when the CONDITION becomes false. The loop may not execute even once if the CONDITION is false initially. These must be a loop terminating condition to avoid infinite looping.

Pseudocode for REPEAT.....UNTIL

```
.....
.....
REPEAT
        Procedure–1
        Procedure–2
        .....
        .....
        .....
        Procedure–n
UNTIL  CONDITION
.....
.....
```

Here, the looping will be executed at least once as the CONDITION is tested in the end of the loop. The loop is executed till the CONDITION remains false and the control comes out of the loop when the CONDITION becomes true. The loop must contain a loop terminating condition to avoid infinite looping.

Examples of Pseudocodes

Example 1. *A company gives the commission to its sales person on the following basis :*

Commission is 1% on the sales less than 1,000

Commission is 5% on the sales greater than or equal to 1,000 but less than 10,000.

Commission is 10% on the sales greater than or above 10,000.

Write the pseudocode for the above commission policy.

Solution : Input SALES

```
IF SALES < 1000 THEN
        COMMISSION = 1% of SALES
```

```
ELSE
        IF SALES < 10000 THEN
                COMMISSION = 5% of SALES
        ELSE
                COMMISSION = 10% of SALES
        ENDIF
ENDIF
Print COMMISSION
```

Example 2. *Write the pseudocode for simulating a simple calculator for performing +, –, *, /.*

Solution :

```
Input two numbers X, Y
Display the menu as
        '+' Add
        '–' Subtract
        '*' Multiply
        '/' Divide
Input the operator (+, –, *, /) as op
IF op = '+' THEN
        Add the two numbers (X + Y) and store the result in SUM
        Print SUM
ENDIF
IF op = '–' THEN
        Subtract the two numbers (X – Y) and store the result in DIFF
        Print DIFF
ENDIF
IF op = '*' THEN
        Multiply the two numbers (X * Y) and store the result in MUL
        Print MUL
ENDIF
IF op = '/' THEN
        IF Y = 0 THEN
                Print "Division by zero not possible"
        ELSE
                Divide the two numbers (X/Y) and store the result in QUO
                Print QUO
        ENDIF
ENDIF
```

Example 3. *Given two dimensional square matrics A and B, each having N rows and columns. Write the pseudocode for storing the product of the matrices in matric C.*

Solution :

```
Input A, B and N
I = 1
DOWHILE I <= N
        J = 1
        DOWHILE J <= N
                SUM = 0
                K = 1
                DOWHILE K <= N
                        SUM = SUM + A[I,K] * B[K,J]
                        Increment the value of K by 1
                ENDDO
                C[I,J] = SUM
                Increment the value of J by 1
        ENDDO
        Increment the value of I by 1
ENDDO
I = 1
DOWHILE I <= N
        J = 1
        DOWHILE J <= N
                PRINT C[I,J]
                Increment the value of J by 1
        ENDDO
        Increment the value of I by 1
ENDDO
```

Here the variables I, J and K are integer variables denoting the array indices. The variable SUM is a dummay variable for storing the intermediate elements of the product matrix C. N denotes the order of the square matrices A and B and that of the resultant matrix C.

Example 4. *Thirty candidates in an engineering examination are required to take 5 single subject papers. For passing the examination, a candidate must score at least 35 marks in each subject and obtain an average of at least 40. The details of each candidate's result is stored in the form of records in a file as given below :*

RollNo Name M1 M2 M3 M4 M5

Write pseudocode to process the file and print the list of successful candidates with their RollNo, Name and total marks.

Solution : Open file

```
Read a record
DOWHILE (NOT END OF FILE)
    Set total = 0
    Set i = 1
    Set failed = false
        DOWHILE (i <= 5 and not failed)
            IF (M_i < 35) THEN
                failed = true
            ELSE
                total = total + M_i
            ENDIF
            Increment the value of i by 1
        ENDDO
        IF failed = false and total >= 200 THEN
            PRINT ROLLNO, NAME, total
        ENDIF
    Read a record
ENDDO
Close file
```

Advantages of Pseudocodes

There are three main advantages of pseudocodes are :

1. The conversion of a pseudocode to a program (in any programming language) is much easier than a flow chart or a decision table.
2. Pseudocode are much easier to modify (when required) as compared to a flowchart.
3. Less time and effort is involved while writing a pseudocode in comparison to draw a flowchart. A programmer can concentrate more on the logic of the program because only fewer rules are to be followed in writing pseudocodes than writing a program in any computer language.

Limitations of Pseudocodes

The limitations of pseudocodes are :

1. For a beginner, it is easier to draw a flowchart to follow the logic of the program in comparison to write the pseudocode for it.
2. A graphical representation of the program logic is not available when we write pseudocode.
3. No standard rules are available in using pseudocode.So, Communication problems exist due to lack of standardization as different programmers follow different styles.

Introduction to Programming

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 32 illustrates this. Once the decision is made to develop a new system, the programmer starts his/her work.

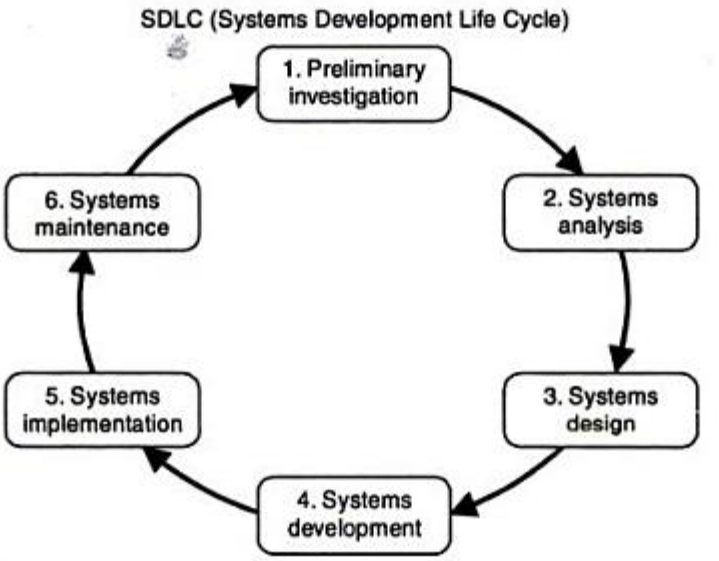


Fig. 32. *Illustration of where programming fits in the SDLC.*

The Phase 4 of the six-phase SDLC includes a five-step procedure of its own as shown in the bottom of Figure 32. 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 in the programming process are given below :

1. Clarify the problem—include needed output, input, processing requirements.
2. Design the program—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. To design the solution, one first needs to write an algorithm. **An algorithm is a formula or set of steps for solving a particular problem.** In computer programming, there are often different algorithms to solve any given problem, and each algorithm has specific advantages and disadvantages in different situations. Inventing elegant algorithms—algorithms that are simple and require the fewest steps possible—is one of the principal challenges in programming.

Algorithms can be expressed in different ways. 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).*

The point of structured programming is to make programs more efficient (with fewer lines of code) and better organized (more redable) and to have better notations so that they have clear and correct descriptions.

The two mini-steps of program design are given below :

(i) Determine the program logic, using a top-down approach

Top-down program design proceeds by identifying the top-element, or module, of a program and then breaking it down in hierarchical fashion to the lowest level of detail. The top-down program design is used to identify the program's processing steps, or modules. After the program is designed, the actual coding proceeds from the bottom up, using the modular approach.

A module is a processing step of a program. Each module is made up of logically related program statements. (Sometimes a module is called a sub-program or subroutine).

Top-down design can be represented graphically in a hierarchy chart. A hierarchy chart, or structure chart, illustrates the overall purpose of the program, by identifying all the modules needed to achieve that purpose and the relationships among them.

(ii) Design details, using pseudocode and/or flowcharts

After determine the essential logic, through the use of top-down programming and hierarchy charts, you can goto work on the details.

There are two ways to show details-write them or draw them; that is, use pseudocode or use flowcharts. Most projects use both methods.

- **Pseudocode :** *Pseudocode is a method of designing a program using normal human-language (an English language) statements to describe the logic and processing flow.* It is much more precise than a written description of a process and much less precise than the actual code. Unlike the final code, pseudocode does not require exact syntax so the programmer is able to focus more on the problem.

  For example,

  ```
  //Pseudocode to calculate monthly bonus
  BEGIN
  ```

```
DO WHILE (so long as) there are employees
    Read an employee record
    Set S to Total Sales
    Set E to Total Expenses
    Net Sales is S – E
    If Net Sales less than 10,000 Then
        Bonus is 0
    Else
        Bonus is 5% of Net Sales
    Endif
    Print Bonus
END DO
END
```

Pseudocode is like an outline or summary form of the program you will write. Sometimes Pseudocode is used simply to express the purpose of a particular programming module in somewhat general terms. With the use of such terms as IF, THEN, or ELSE, however, the Pseudocode follows the rules of *control structures*.

- **Program flowcharts :** *A program flowchart is a chart that graphically presents the detailed series of steps (algorithm or logical flow) needed to solve a programming problem.*
- **Control structures :** *A control structure or logic structure, is a structure that controls the logical sequence in which computer program instructions are executed. In structured program design, three control structures are used to form the logic of a program : sequence, selection and iteration (or loop).*

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.

Identifying arithmetic and logical operations required for solutions and using appropriate control structures such as conditional or looping control structure is very important.

(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 for giving 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.* These days manuals are usually on the software CD.

(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.

Table 2 summarizes the five steps of the programming process.

Table 2. Five Steps of Programming Process

Step 1 : Problem definition	1. Specify program objectives and program users. 2. Specify output requirements. 3. Specify input requirements. 4. Specify processing requirements. 5. Study feasibility of implementing program. 6. Document the analysis.
Step 2 : Program design	1. Determine program logic through top-down approach and modularization, using a hierarchy chart. 2. Design details using pseudocode and/or using flowcharts, preferably on the basis of control structures. 3. Test design with structured walkthrough.
Step 3 : Program coding	1. Select the appropriate high-level programming language. 2. Code the program in that language, following the syntax carefully.
Step 4 : Program testing	1. Desk-check the program to discover errors. 2. Run the program and debug it (alpha testing). 3. Run real-world data (beta testing).
Step 5 : Program documentation and maintenance	1. Prepare user documentation. 2. Write operator documentation. 3. Write programmer documentation. 4. Maintain the program.

Five Generations of Programming Languages

A *programming language* is a set of rules that tells the computer what operations to do. These languages are used by the programmers to create other kinds of software. Many programming languages have been written to solve particular kinds of problems. These languages have one thing in common, these must be reduced to digital form—a 1 or 0, electricity on or off—because that is all the computer can work with.

To see how it works, this is important to understand that there are five levels, or generations, of programming languages, ranging from low-level to high-level. The five generation of programming languages start at the lowest level with (1) machine language. They then range up through (2) assembly language, (3) high-level languages (procedural and object-oriented languages), and (4) very high-level languages (problem-oriented languages). At the highest level are (5) natural languages.

Programming languages are said to be *lower level* when they are closer to the language that the computer itself uses—the 1s and 0s. They are called *higher level* when they are closer to the language people use more like English, for example. Beginning in 1945, the five levels or generations have evolved over the years, as programmers gradually adopted the later generations.

The five generations of programming languages is given in Table 3.

Table 3. The five generations of programming languages

First-Generation Language	More technical
machine language	More flexible
Second-Generation Language	Less user friendly
assembly language	Faster
Third-Generation Language	Less technical
COBOL, BASIC, C, Ada	Less flexible
compilers, interpreters	More user friendly
Fourth-Generation Language	Slower
report generators	
query languages	
application generators	
Fifth-Generation Language	Current and future
natural language	development

The births of the generations are given below :

- First generation, 1945—*Machine language.*
- Second generation, mid-1950s—*Assembly language.*
- Third generation, mid-1950s to early 1960s—*High-level languages (procedural languages and object-oriented)* : For example, COBOL, BASIC, C and C++.

- Fourth generation, early 1970s—*Very high-level languages (problem oriented languages)* : For example, SQL, Intellect, NOMAD, FOCUS.
- Fifth generation, early 1980s—*Natural languages.*

Let us discuss these five generations :

First Generation : Machine Language

Machine language is the basic language of the computer, representing data as 1s and 0s. Each CPU model has its own machine language. Machine language programs vary from computer to computer, *i.e.,* they are *machine-dependent.* These binary digits, which correspond to the on and off electrical states of the computer, are clearly not convenient for people to read and use. Believe it or not, though, programmers *did* work with these mind- numbing digits. There must have been great signals of relief when the next generation of programming languages—assembly language—came along.

Second Generation : Assembly Language

Assembly language is a low-level programming language that allows a computer user to write a program using abbreviations or more easily remembered words instead of numbers. A programmer can write instructions in assembly language more quickly than in machine language. In these languages, each numeric instruction is assigned a short name (called a mnemonic) that is easier to remember than a number. Assembly languages are still widely used in some situations, where execution times are critical. Moreover, assembly language has the same drawback as machine language in that it varies from computer to computer—it is machine-dependent.

A computer can execute programs only in machine language, a translator or converter is required if the program is written in any other language. *A language translator is a type of systems software that translates a program written in a second, third or higher generation language into machine language.*

Language translators are of three types :

- Assemblers
- Compilers
- Interpreters.

An assembler, or assembler program, is a program that translates the assembly-language program into machine language.

Third Generation : High-Level or Procedural Languages

A high-level or procedural language resembles some human language such as English. For example, COBOL, which is used for business applications. A procedural language allow users to write in a familiar notation, rather than numbers or abbreviations. Also, unlike machine and assembly languages, most of procedural languages are not machine—dependent—*i.e.,* they can be used on more than one kind of computer. Few examples, are FORTRAN, COBOL, BASIC, Pascal and C.

For a procedural language we need *language translator* to translate it into machine language. Depending on the procedural language we may use either of the following types of translators :

a *compiler* or *an interpreter.*

• *Compiler—execute later. A compiler is a language translator that converts the entire program of a high-level language into machine language before the computer executes the program.* The programming instructions of a procedural language are called the **source code.** The compiler translates it into machine language, which in this case is called the **object code.** The important point here is that the object code can be saved and thus can be executed later (as many times as desired), rather than run right away. The executable files—the output of compilers—have the **.exe** extension.

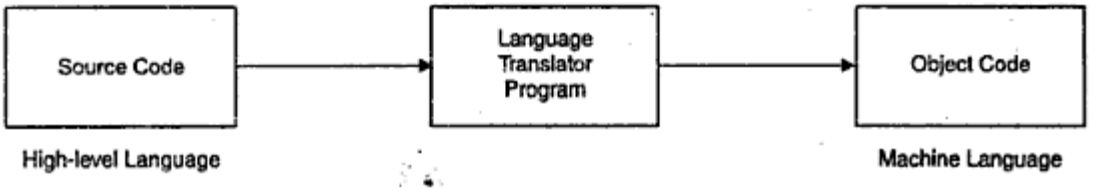


Fig. 33.

Examples of procedural languages using compilers are COBOL, FORTRAN, Pascal and C.

• *Interpreter—execute immediately. An interpreter is a language translator that converts each procedural language statement into machine language and executes it immediately, statement by statement.* In contrast to the compiler, no object code is saved. Therefore, interpreted code generally runs slower than compiled code. However, code can be tested line by line. BASIC language uses an interpreter.

Who cares, you might say, whether you can run a program now or later ? (After all, "later" could be only a matter of seconds or minutes.) Here is the significance : When a compiler is used, it requires two steps (the source code and the subject code) before the program can be executed. The interpreter, on the other hand, requires only *one* step.

The advantage of a compiler language is that, once we get the object code, the program runs faster. The advantage of an interpreted language, on the other hand, is that programs are easier to develop. Some language translators—such as those with C++ and Java, can both compile and interpret.

Some of the most popular procedural languages are Visual BASIC, C and C++.

Fourth Generation : Very High-Level or Problem Oriented Languages

Third-generation languages tell the computer how to do something. Fourth generation languages, in contrast, tell the computer what to do. *Very high-level or problem-oriented languages, also called fourth-generation languages (4 GLs), are much more user-oriented and allow users to develop programs with fewer commands compared with procedural languages,* although they require more computing power. These languages are known as *problem-oriented* because they are designed to solve specific problems, whereas procedural languages are more general-purpose languages.

Three types of problem-oriented languages are report generators, query languages, and application generators.

Report generators

A report generator, also known as a report writer, is a program for end-users that produces a report. The report may be a printout or a screen display. It may show all or part of a database file. We can specify the format in advance-columns, headings, etc., and the report generator will then produce data in that format.

Report generators (an example is RPG III) were the precursor to today's query languages.

Query languages

A query language is an easy-to-use language for retrieving data from a database management system. The query may be given in the form of a sentence or near-English command. Or the query may be obtained from choices on a menu.

Examples of query languages are SQL (for Structured Query Language) and Intellect. For example, with Intellect, which is used with IBM mainframes, you can ask an English-language question such as "Tell me the number of students in the computer department."

Application generators

An application generator is a programmer's tool consisting of modules that have been preprogrammed to accomplish various tasks. The benefit is that the programmer can generate applications programs from descriptions of the problem rather than by traditional programming, in which he/she has to specify how the data should be processed.

Application generators are used by the programmers to create parts of other programs. For example, the software is used to construct on screen menus or types of input and output screen formats. NOMAD and FOCUS, two database management systems, include application generators.

Fifth Generation : Natural Languages

Natural languages are of two types. The first are ordinary human languages : English, Spanish etc. The second are programming languages that use human language to provide people a more natural connection with computers.

Natural languages are part of the field of study known as *artificial intelligence.* Artificial intelligence (AI) is a group of related technologies that attempt to develop machines capable of emulating human-like qualities, such as learning, reasoning, communicating, seeing and hearing.

Programming Languages used Today

Let us now consider some of the third-generation, or high-level languages in use today.

BASIC : BASIC is the easy language. BASIC was developed by John Kemeny and Thomas Kurtch in 1965 for use in training their students at Dartmouth College. By the late 1960s, it was widely used in academic settings on all kinds of computers, from mainframes to personal computers (PCs).

BASIC (Beginner's All-purpose Symbolic Instruction Code) used to be the most popular microcomputer language and is considered the easiest programming language to learn. Although it is available in compiler form, the interpreter form is more popular with first-time and casual users. This is because it is interactive, meaning that user and computer can communicate with each other during the writing and executing (running) of the program. Today there is no one version of BASIC. One of the popular current evolutions is Visual BASIC.

The advantage and disadvantages of BASIC are given below :

- **Advantage :** The primary advantage of BASIC is its ease of use.
- **Disadvantages :** (1) Its processing speed is relatively slow, although compiler versions are faster than interpreter versions.

(2) There is no one version of BASIC, although in 1987 ANSI adopted a new standard that eliminated portability problems—that is, problems with running it on different computers.

COBOL : COBOL is the language of business and it was formally adopted in 1960, *COBOL (for Common Business-Oriented Language) is the most frequently used business programming language for large computers.* Its most significant attribute is that it is extremely readable. For example, a COBOL statement might be :

MULTIPLY HOURLY-RATE BY HOURS-WORKED GIVING TOTAL-PAY

Writing a COBOL program resembles writing an outline for a research paper. The program is divided into four divisions :

IDENTIFICATION DIVISION

ENVIRONMENT DIVISION

DATA DIVISION

PROCEDURE DIVISION

The divisions in turn are divided into sections, which are divided into paragraphs, which are further divided into sections. The IDENTIFICATION DIVISION identifies the name of the program and the author (programmer) and perhaps some other helpful comments. The ENVIRONMENT DIVISION describes the computer on which the program will be compiled and executed. The DATA DIVISION describes what data will be processed. The PROCEDURE DIVISION describes the actual processing procedures.

COBOL, too, has both advantages and disadvantages.

- **Advantages:** (1) It is machine independent.
  (2) Its English-like statements are easy to understand, even for a nonprogrammer.
  (3) It can handle many files, records, and fields.
  (4) It easily handles input/output operations.
- **Disadvantages:** (1) Because it is so readable, it is wordy. Thus, even simple programs are lengthy, and programmer productivity is slowed.
  (2) It cannot handle mathematical processing as well as FORTRAN.

FORTRAN : FORTRAN is the language of mathematics and the first high-level language. It was developed in 1954 by IBM, ***FORTRAN* (for *FORmula TRANslator*) was the first high-level language.** Originally designed to express mathematical formulas, it is still the most widely used language for mathematical, scientific, and engineering problems. It is also useful for complex business applications, such as forecasting and modeling. However, because it cannot handle a large volume of input/output operations or file processing, it is not used for more typical business problems.

FORTRAN has both advantages and disadvantages:

- **Advantages :** (1) FORTRAN can handle complex mathematical and logical expressions.
  (2) Its statements are relatively short and simple.
  (3) FORTRAN programs developed on one type of computer can often be easily modified to work on other types.
- **Disadvantages :** (1) FORTRAN does not handle input and output operations to storage devices as efficiently as some other higher-level languages.

(2) It has only a limited ability to express and process non-numeric data.

(3) It is not as easy to read and understand as some other high-level languages.

Pascal : Pascal is the simple language. Named after the 17th-century French mathematician Blaise Pascal, *Pascal is an alternative to BASIC as a language for teaching purposes and is relatively easy to learn.* A difference form BASIC is that Pascal uses structured programming.

A compiled language, Pascal offers these advantages and disadvantages:

- **Advantages :** (1) Pascal is easy to learn.

  (2) It has extensive capabilities for graphics programming.

  (3) It is excellent for scientific use.

- **Disadvantage :** Pascal has limited input/output programming capabilities, which limits its business applications.

C : C is the language for portability and scientific purpose. "C" is the language's entire name, and it does not "stand" for anything. Developed at Bell Laboratories in the early 1970s, *C is a general-purpose, compiled language that works well for microcomputers and is portable among many computers.* It was originally developed for writing system software. (Most of the Unix operating system was written using C.) Today it is widely used for writing applications, including word processing, spreadsheets, games, robotics, and graphics programs. It is now considered a necessary language for programmers to know.

Here are the advantages and disadvantages of C:

- **Advantages :** (1) C works well with microcomputers.

  (2) It has a high degree of portability—it can be run without change on a variety of computers.

  (3) It is fast and efficient.

  (4) It enables the programmer to manipulate individual bits in main memory.

- **Disadvantages :** (1) C is considered difficult to learn.

  (2) Because of its conciseness, the code can be difficult to follow.

  (3) It is not suited to applications that require a lot of report formatting and data file manipulation.

C++ : C++ is an Object Oriented Programming (OOP) language. In *C++—the plus signs stand for "more than C"—which combines the traditional C programming language with object-oriented capability.* C++ was created by *Bjarne Stroustrup.* With C++, programmers can write standard code in C without the object-oriented features, use object-oriented features, or do a mixture of both.

Three important concepts of OOP are :

Encapsulation

Inheritance

Polymorphism

Here are the advantages and disadvantages of C++ :

- **Advantages :** (1) C++ works well with microcomputers.

  (2) It is portable.

(3) Once the programmer has written a block of program code, it can be reused in any number of program, *i.e.,* you don't have to start from scratch.

(4) In OOP an object can be used repeatedly in different applications and by different programmers, speeding up development time and lowering costs.

- **Disadvantage :** It takes longer to learn than traditional programming because it means training oneself to a new way of thinking.

GUI based Languages

The Graphical User Interface (GUI) as the name suggests, uses illustrations for text, which help users to interact with an application. This feature makes it easier to understand things in a quicker and easier way.

Coding (writing program) in GUI environment is quite a transition to traditional, linear programming methods where the user is guided through a linear path of execution and is limited to a small set of operations. Using GUI environment, the number of options available to the user is much greater, permitting more freedom to the user and the developer.

Essentially, visual programming takes OOP (Object Oriented Programming) to the next level. The goal of visual programming is to make programming easier for programmers and more accessible to nonprogrammers, by borrowing the object orientation of OOP languages but exercising it in a graphical or visual way. Visual programming helps users to think more about the problem solving than about handling the programming language. There is no learning of syntax or actual writing of code.

Visual programming is a method of creating programs in which the programmer makes connections between objects by drawing, pointing, and clicking on diagrams and icons and interacting with flowcharts. Thus, the programmer can create programs by clicking on icons that represent common programming routines.

Visual Basic : Visual Basic is an example of visual programming. *Visual BASIC is a Windows-based, object-oriented programming language from Microsoft that lets users develop Windows and office applications by*

(1) creating command buttons, text boxes, windows, and toolbars, which

(2) then may be linked to small BASIC programs that perform certain actions. Visual BASIC is "event-driven," which means that the program waits for the user to do something (an "event"), such as click on an icon, and then the program responds. At the beginning, for example, the user can use drag-and-drop tools to develop a graphical user interface, which is created automatically by the program. As Visual BASIC is easy to use so it allows even novice programmers to create impressive Windows-based applications.

Visual C++ : This language is a GUI extension of conventional C++ language. It is a part of *Microsoft Visual Studio* software package. It is an Object Oriented Programming language that has been designed for producing high level object oriented applications, that can work with hardware devices, for example—Windows applications and device drivers.

Visual C++ requires a good knowledge of C++ and Windows architecture. It contains many built in functions and classes for common objectives, under Windows platform.

C# (C Sharp) : Object-oriented concepts form the base of all modern programming languages. Understanding the basic concepts of object-orientation helps a programmer or developer to use various modern day programming languages, more effectively. **C#,** also known as **C-Sharp,** is an object-oriented programming language developed by *Microsoft* that intends to be a simple, modern, and general-purpose programming language for application development. C# is a case-sensitive language.

C# contains features similar to Java and C++. It implements object-oriented concepts, such as *abstraction, encapsulation, polymorphism, and inheritence.* C# also has various programming concepts, such as *threads, file handling, delegates, attributes, and reflection.* It is specially designed to work with **Microsoft's .NET** platform.

> ***Note :*** *The .Net platform is aimed at providing Internet users with Web-enabled interface for applications and computing devices such as mobile phones. It also provides programmers or developers the ability to create reusable modules, thereby increasing productivity.*

Java : Java is a new procedural language that has gained great popularity. It is the language used for creating interactive Web Pages. It is developed by James Gosling and available from Sun Microsystems. Originally Java was called Oak. Derived from C++, Java is a major departure from the HTML (Hypertext Markup Language) coding that makes up most web pages. Sitting atop markup languages such as HTML and XML (eXtensible Markup Language), *Java is an Object-Oriented Programming (OOP) language allowing programmers to build applications that can run on any operating system.* With Java, big applications programs can be broken into mini-applications, or "applets," that can be downloaded off the Internet and execute (run) on any computer. Like C++, Java is also an Object Oriented Programming language but there are some differences as well. In Java there are :

- No pointers
- No multiple inheritance
- No goto statement
- No operator overloading.

In C++, we have *new* and *delete* operators for memory allocation and deallocation. In Java we have an embedded auto garbage collection mechanism. The memory area which is not required is automatically made free by the garbage collector in Java language. So Java is simple in comparison to C++. Java code is divided into classes. Java resembles a client/server model. It is reliable (as it permits the programmer to write programs that do not crash when least expected and will also be bug-free). It is comparable in speed with C++. Java has sophisticated multitasking features which are integrated into the language itself, which make it very simple to use and, at the same time very robust (Robustness means the program should not hang but terminate with an appropriate error message when wrong data is provided by the user).

Vb.net : As mentioned earlier the .NET platform is aimed at providing Internet users with Web-enabled interface for applications and computing devices such as mobile phones. It also provides programmers or developers the ability to create reusable modules, thereby increasing productivity.

VB.NET is the next generation of Visual Basic and it is designed to be the easiest and most productive tool for creating .NET applications, including Windows applications, Web

Services, and Web applications. It is a powerful Object Oriented Programming language. Some features of Visual Basic.NET are :

- All Features of Visual Basic
- Inheritance
- Method Overloading
- Structured Exception Handling
- Free Threading

The most significant aspect of .NET architecture is that code in Visual Basic and C++ is compiled not to native executable code, but to an *Intermediate Language (IL)*, with the final step of converting to native executable code normally happening at runtime. Such code is called *managed code*. C++ code can optionally be marked as managed code, in which case it gets compiled to *Intermediate Language* too. The *multiple inheritance* feature is restricted in VB.NET as it is not supported on .NET.

Summary

- A program is a sequence of instructions written in a programming language.
- For better designing of a program, a systematic planning must be done.
- Computers work because they are fed a series of step-by-step instructions called programs.
- An algorithm is a step-by-step procedure to solve a problem in unambiguous finite number of steps. An algorithm is independent of any programming language.
- A flowchart is a graphical way of illustrating the steps in a process. It uses symbols connected by flowlines to represent processes and the direction of flow within the program. A flowchart is independent of any programming language.
- Some useful techniques of problem solving other than flowcharting are modular programming, top-down design and structured programming.
- A good program has the characteristics, efficiency, flexibility, reliability, portability and robustness etc.
- Modular programming is breaking down of a problem into smaller independent pieces (modules).
- Computer problem solving is about understanding.
- The preliminary investigation may be thought of as the problem definition phase.
- The old computer proverb states, "the sooner you start coding your program the longer it is going to take".
- Probably the most widely known and most often used principle for problem solving is the *divide-and-conquer* strategy. It is widely used with searching, selection and sorting algorithms.
- The main objectives of structured programming are readability, clarity of programs, easy modification and reduced testing problems.
- Top-down programming is also known as the *process of stepwise refinement.*

- Bottom-up programming approach is the reverse of the top-down programming.
- Pseudocode is a method of designing a program using normal human-language (an English language) statements to describe the logic and processing flow.
- Programming is a multistep process for creating the list of instructions.
- The five steps in the programming process are :
  1. Clarify the problem.
  2. Design the program.
  3. Code the program.
  4. Test the program.
  5. Document and maintain the program.
- A programming language is a set of rules that tells the computer what operations to do. The five generations of programming languages are machine language, assembly language, high-level languages, very high-level languages and natural languages.

Review Questions and Exercises

1. Write a short note on planning the computer program.
2. Describe the various problem solving techniques.
3. Explain in detail the various characteristics of a good program.
4. What is modular approach ? Explain.
5. What is step-wise refinement ? Explain by giving a suitable example.
6. Explain the use of program control structures in problem solving with suitable examples.
7. Explain the concept of structured programming.
8. What is an algorithm ? Explain its need.
9. What are the characteristics necessary for a sequence of instructions to qualify as an algorithm ?
10. What are flowcharts ? How do they help in problem solving ? Illustrate.
11. What is a flowchart ? List the flowcharting rules.
12. Write an algorithm and draw a flowchart to check an integer for even or odd.
13. Draw a flowchart and write the algorithm for finding the smallest of three numbers.
14. Draw a flowchart to accept marks of students having *n* subjects, find their average and calculate the highest marks in the class.
15. Draw a flowchart and write the algorithm to find the smallest of N numbers.
16. Draw a flowchart and write the algorithm to find the GCD (Greatest Common Divisor) of two numbers.
17. Draw a flowchart and write the algorithm for factorial computation.
18. Draw a flowchart and write an algorithm for checking a number of palindrome.
19. Draw a flowchart and write the algorithm for printing the fibonacci terms upto a specific limit.

20. Draw a flowchart and write the algorithm for printing first N prime numbers.
21. Draw a flowchart for searching an element from a sorted array (ascending order) having N elements using binary search method.
22. Draw a flowchart for finding the transpose of a matrix A of order N × N.
23. Write an algorithm and draw a flowchart for addition of two matrices.
24. Write an algorithm and draw a flowchart to print elements of upper triangular matrix.
25. Draw a flowchart and write the algorithm for checking a square matrix for symmetry.
26. Is it feasible to have more than one flowchart for a given problem ? Give reasons for your answer.
27. Differentiate between a macroflowchart and a microflowchart. Give examples.
28. What are the various guidelines to be followed while drawing a flowchart ? Discuss the advantages and limitations of flowcharting.
29. What is a pseudocode ? Why is it so called ? Give another name for pseudocode.
30. What is indentation ? What is its importance in writing pseudocodes ?
31. Write the pseudocode to produce a printed listing of all students over the age 18 in a class. The input records contain the name and age of the students. Assume a sentinel value of 111 for the age field of the trailer record.
32. Each paper in a set of examination papers includes a grade of A, B, C, D or E. A count is to be made of how many papers have the grade of B and how many have the grade of D ? The total count of both types have to be printed at the end. Prepare a flowchart to perform this task. Assume a suitable sentinel value for the trailer record. Write the pseudocode also.
33. Write the pseudocode to find the sum of all the odd numbers between 0 and 200. Before ending, print the result of the calculation.
34. Write the pseudocode for checking the type of a triangle. Assume that the angles of the triangle are given as input. Print the answer as yes or no.
35. Discuss the advantages and limitations of pseudocode.
36. How are programming needs clarified ?
37. How is a program designed ?
38. What is involved in coding a program ?
39. How is a program tested ?
40. What is involved in documenting and maintaining a program ?
41. What is programming, and what are the five steps in accomplishing it ?
42. State the steps typically followed for the in-house development of a software package.
43. What are the five generations of programming languages ?
44. What are some third-generation languages, and what are they used for ?
45. How do OOP and visual programming work ?

❑❑❑

Overview of C

Introduction

"C" is the language's entire name, and it does not "stand" for anything. C is the successor of B, which was the successor of BCPL, which was the successor of CPL (Computer Programming Language), an early programming language that was not implemented. Developed at Bell Laboratories in the early 1970's by Dennis Ritchie, C is a general-purpose, compiled language that works well for microcomputers and is portable among many computers. It was originally developed for writing system software. The first major program written in C was the Unix operating system, and for many years C was considered to be inextricably linked with Unix. Now, however, C is an important language independent of Unix. Today it is widely used for writing applications, including word processing, spreadsheets, games, robotics, and graphics programs. It is now considered a necessary language for programmers to know.

C is often called a *middle-level* computer language. It does not mean that C is less powerful, harder to use, or less developed than a high-level language such as BASIC or Pascal; nor does it imply that C has the cumbersome (difficult to understand) nature of assembly language. Rather, C is thought of as a middle-level language because it combines the best elements of high-level languages with the control and flexibility of assembly language. Table 1 shows how C fits into the spectrum of computer languages.

Table 1. C's Place in the World of Computer Languages

Highest level	Ada Modula-2 Pascal COBOL FORTRAN BASIC
Middle level	Java C# C++ C Forth
Lowest level	Macro-assembler Assembler

C is commonly referred to simply as a *structured* language. *The reason that C is not, technically, a block-structured language is that block-structured languages permit procedures or functions to be declared inside other procedures of functions. However, since C does not allow the creation of functions within functions, it cannot formally be called block-structured.*

A structured language allows you a variety a programming possibilities. It directly supports several loop constructs, such as **while, do-while,** and **for**. In a structured language, the use of **goto** is either prohibited or discouraged and is not the common form of program control. A structured language allows you to place statements anywhere on a line and does not require a strict field concept.

Some examples of structured and nonstructured languages are given below :

Nonstructured	Structured
FORTRAN	Pascal
BASIC	Ada
COBOL	Java
	C#
	C++
	C
	Modula-2

C's main structured component is the function—C's stand-alone subroutine. In C, functions are the building blocks in which all program activity occurs. Another way to structure code in C is through the use of code blocks. A *code block* is a logically connected group of program statements that is treated as a unit. Code blocks allow many algorithms to be implemented with clarity, elegance, and efficiency. Moreover, they help the programmer better conceptulize the true nature of algorithm being implemented.

The advantages and disadvantages of C are given below :

Advantages :

1. C works well with microcomputers.
2. It has a high degree of portability—it can run on a variety of computers.
3. It is fast and efficient.
4. It enables the programmer to manipulate individual bits in main memory.
5. It requires less memory than many other programming languages.

The advantages written above are also the important characteristics of C language.

Disadvantages :

1. C is considered difficult to learn.
2. Because of its conciseness, the code can be difficult to follow.
3. It is not suited to applications that require a lot of report formatting and data file manipulation.

For learning how to write programs in C, we must first know what alphabets, numbers and symbols are used, then how using them constants, variables and keywords are formed, and finally how are these combined to form an instruction. A program is written using a group of instructions. Figure 1 illustrates this :

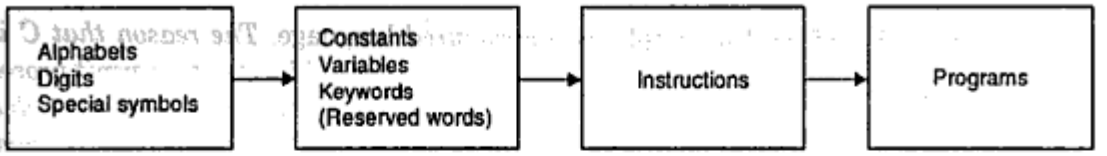


Fig. 1. *Systematic way of learning C language.*

C Character Set

A programming language processes some kind of data and provides some meaningful result known as information. The data and information are represented by the character set of the language. A sequence of finite instructions is written following the syntax rules (or grammar) called the program, for getting the desired result. Every program instruction must be coded precisely and user friendly way to help the programmer and others. C language has its own character set for coding compact and efficient programs.

A character represents any alphabet, digit or special character used to form words, numbers and expressions. The C character set can be divided into the following groups :

Alphabets or Letters : Capital A to Z
Small a to z

Digits : All decimal digits 0 to 9

White Spaces : Blank space
Horizontal tab
Vertical tab
Newline
Form feed

Special Characters

The special characters are given in Table 2.

Table 2. Special Characters

,	Comma	&	Ampersand
.	Period or dot	^	Caret
;	Semicolon	*	Asterisk
:	Colon	-	Minus
'	Apostrophe	+	Plus
"	Quotation mark	<	Less than
!	Exclamation mark	>	Greater than
|	Vertical bar	( )	Parenthesis left/right
/	Slash	{ }	Braces left/right

\	Back slash	[ ]	Brackets left/right
~	Tilde	%	Percent
_	Underscore	#	Hash or number sign
$	Dollar	=	Equal to
?	Question mark	@	At the rate

The white space characters are ignored by the C compiler in cases where they are not the part of a string constant. We can use white spaces for separation of words but these are not allowed between the characters of reserved words (keywords) and identifiers (name of some program element).

C Tokens

The smallest individual units of a C program are known as tokens. We can code a C program using the tokens and its syntax rules. Figure 2 shows the six types of C tokens:

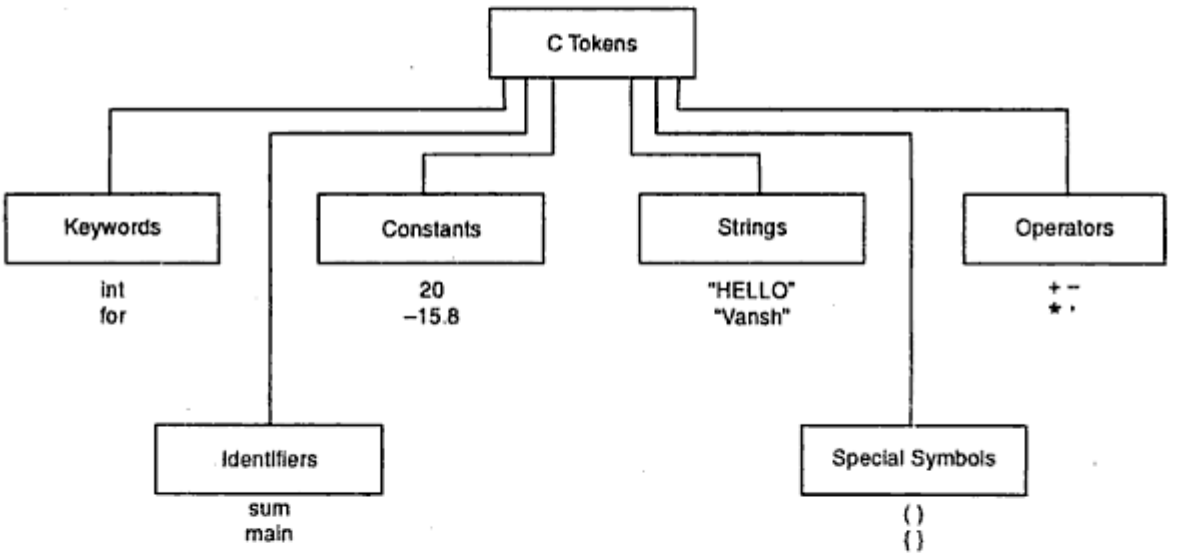


Fig. 2. *C tokens with examples.*

Keywords and Identifiers

A word in C is named either a keyword or an identifier. **Keywords** are those words whose meaning has been fixed and is known to the compiler. We cannot change the meaning of the keywords, if we try to do so an error message will occur. Keywords or Reserved words form the basic building block in the formation of C statements. Table 3 shows the keywords in C.

Table 3. Keywords or Reserved Words

auto	double	if	static
break	else	int	struct
case	enum	long	switch
char	extern	near	typedef
const	float	register	union
continue	far	return	unsigned
default	for	short	void
do	goto	signed	while

The **C keywords must be written in lowercase.** In addition to the above reserved words some other reserved words may also be available with your compiler which can be consulted from the C user manual.

Identifiers are used for naming program elements *i.e.,* variables, symbolic constants, functions and arrays. These are defined by the user. The user should try to give meaningful names while using identifiers so that the program can be understood with minimum difficulty. The following rules should be followed when an identifier is named :

(*i*) An identifier must begin with a letter and must be followed by letters or digits.

(*ii*) Underscore ('_') is permitted and considered as a letter. It is used for better understanding of the identifier name.

(*iii*) Name of an identifier cannot be anyone out of the reserved words.

(*iv*) The **C language is case sensitive** (*i.e.,* uppercase and lowercase letters are considered different). Thus the names sum, *SUM* and *sum* are all different identifiers. Generally lower case letters are used for variable names, function names and uppercase letters for symbolic constants.

(*v*) The ANSI (American National Standard Institute) C compiler allows at least 31 characters in an identifier name. It is a good practice to have identifiers with few letters; up to 8 letters is generally preferred.

Constants

In C, all the data processed by a program is stored either as a variable or a constant. Constants are the data items that never change their value during the program execution. C constants can be divided into two major categories as shown in Figure 3.

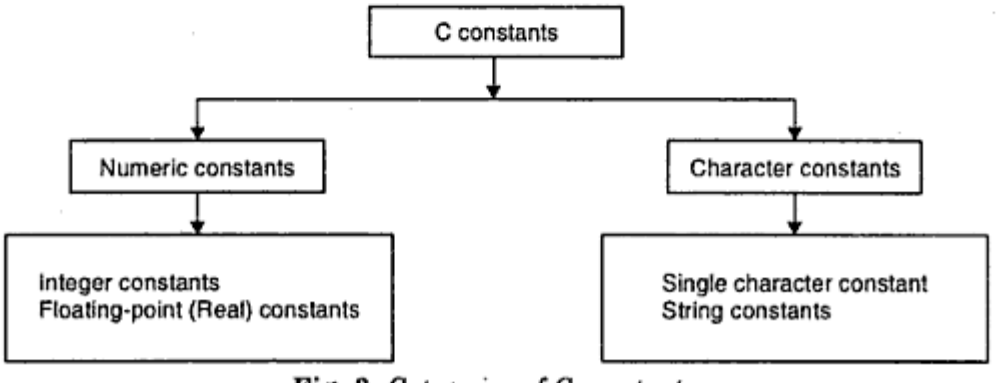


Fig. 3. *Categories of C constants.*

Let us discuss some C constants which are easy to follow at this stage :

(*i*) Integer constants

(*ii*) Floating-point constants

(*iii*) Character constants

(*iv*) Back slash constants

(*v*) String constants

Integer Constants

These are whole numbers without any fractional part. The following rules are followed for constructing integer constants :

(*a*) It must have at least one digit.

(*b*) It must not contain a decimal point.

(*c*) It may either have + or – sign.

(*d*) When no sign is present it is assumed to be positive.

(*e*) Commas and blanks are not permitted in it.

In C, there can be three types of integer constants :

Decimal (base or radix 10)

Octal (base or radix 8)

Hexadecimal (base or radix 16).

Decimal integer constants. These consist of a sequence of digits, 0 through 9.

The following are valid integer constants :

1024 3313 275 – 12 32767 – 5000

The following are invalid integer constants :

.135 (Decimal point not permitted)

2,450 (Comma not permitted)

51237 (Value larger than permitted range)

Range of integer constants. The computer memory consists of a number of cells called words. The number of bits in a word is called wordlength. Generally the PC's have wordlength of 16 bits (some may have 32 or more). Let us take n as the wordlength. Then the integer range is given by -2^{n-1} to $2^{n-1} - 1$.

If $n = 16$, the integer range is -2^{16-1} to $2^{16-1} - 1$.

i.e., – 32768 to 32767.

If an integer less than – 32768 is given, an underflow error will occur and if the integer is more than 32767 an overflow error will occur.

Table 4 illustrates the types of integers with their range and format for 16-bit word-length :

Table 4. Integer Range

Octal integer constants : These are preceded by 0 (digit zero) and consist of sequence of digits 0 through 7. For example, decimal integer 14 will be written as 016 as octal integer (as $14_{10} = 16_8$). The following are some valid octal integer constants :

0235
047
0

Hexadecimal integer constants : These are preceded by 0*x* or 0*X* and consist of digits 0 through 9 and may also include alphabets A through *F* or a through *f*. The letters A through *F* denote the numbers 10 through 15. For example, decimal integer 14 will be written as 0*XE* as hexadecimal integer (as $14_{10} = E_{16}$).

The following are some valid hexadecimal integer constants :

0*X*7
0*x*5*F*
0*X*a*ef*

> ***Note :*** *Octal and hexadecimal numbers are rarely used in programming.*

The suffix *l* or *L*, *u* or *U* and *ul* or *UL* allow any constant to be represented as long, unsigned or unsigned long respectively.

Floating-point Constants (Real Constants)

These have fractional parts to represent quantities like average, height, area etc., which cannot be represented by integer numbers precisely.

These may be written in either **fractional form** or **exponent form.**

A real constant could be written in the following form :

[sign] [integer] · [fraction] [exponent]

where the integer part or the fractional part may be omitted but not both.

The following rules are followed for constructing real constants in fractional form :

(*a*) A floating-point constant in fractional form must have at least one digit before and after the decimal point.

(*b*) It may either have + or – sign.

(*c*) When no sign is present it is assumed to be positive.

(*d*) Commas and blanks are not permitted in it.

The following are valid real constants in fractional form :

12.5 – 15.8 – 0.0055 336.0

The following are invalid real constants in fractional form :

125 (Decimal point missing)

5,415.6 (Comma not allowed)

The exponent form consists of two parts : **mantissa** and **exponent.** These are usually used when the constant is either too small or too big. But, there is no restriction for us to use exponential form for other floating constants.

In exponent form the part before 'e' is called mantissa and the part after 'e' is called exponent. We can write **e** or **E** for separating the mantissa and exponent. Since the decimal point can "float" due to use of exponent, the number represented in this form gives the floating-point representation. The following rules are followed for constructing real constants in exponent form :

(*a*) The mantissa and exponent are separated by **e.**

(*b*) The mantissa must be either an integer or a proper real constant.

(*c*) The mantissa may have either + or – sign.

(*d*) When no sign is present it is assumed to be positive.

(*e*) The exponent must be at least one digit integer (either positive or negative). Default sign is +.

The following are valid real constants in exponent form :

+ 15.8E5 .05E – 3

The following are invalid real constants in exponent form :

– 125.8 E (No digit specified for exponent)

15.7 E 2.5 (Exponent cannot be fraction)

Range of floating constants

The range of floating constants in exponent form on a 16-bit PC is –3.4e38 to 3.4e38. It occupies 4 bytes of memory. This range is for single precision real numbers. The precision of this data type is seven decimal digits. Whereas the double precision value occupies 8 bytes of memory, range is from –1.7e308 to 1.7e308 and the precision is fourteen decimal digits.

Character Constants

These consist of a single character enclosed by a pair of single quotation marks. A character value occupies 1-byte of memory. A character constant cannot be of length more than 1. *For example,*

'A' '9' ';' ' '

Here the last constant represents a blank space. The character constant '9' is different from the number 9. Each character constant has an ASCII value associated with it. For example, the following statements will print 65 and A respectively :

```
printf("%d", 'A');
printf("%c", '65').
```

Arithmetic operations on characters are possible due to the fact that each character constant associates an integer value with it.

Back Slash Constants or Escape Sequences

These character constants represent one character, although they consist of two characters. These are also known as escape sequences. These are interpreted at execution time. The values of these characters are implementation-defined.

C uses some characters such as line feed, form feed, tab, newline etc., through execution characters *i.e.*, which cannot be printed or displayed directly. Table 5 shows some of the escape sequence characters or back slash character constants :

Table 5. Escape Sequences

Escape sequence	*Meaning*	*Execution time result*
'\0'	End of string	NULL
'\n'	End of line	Takes the control to next line
'\r'	Carriage return	Takes the control to next paragraph
'\f'	Form feed	Takes the control to next logical page
'\t'	Horizontal tab	Takes the control to next horizontal tabulation position
'\v'	Vertical tab	Takes the control to next vertical tabulation position
'\b'	Back space	Takes the control to the previous position in the current line
'\\'	Back slash	Presents with a back slash \
'\a'	Alert	Provides an audible alert
'\" '	Double quote	Presents with a double quote

String Constants

These consist of a sequence of characters enclosed in double quotes. The characters enclosed in double quotes can be alphabets, digits, blank space and special characters. *For example,*

"Hello ! World"
"Year 2012"
"Sum"
"A"

Remember that a character constant (*e.g.*, 'A') is not equivalent to the single character string constant (*e.g.*, "A") in C language. As mentioned earlier the equivalent integer value for 'A' is 65 but "A" does not have any such value. We use strings in C programs for providing suitable messages in output statements. Character strings are quite useful in certain situations and are discussed later on.

Meaning of Variables

In C, a *variable* is an entity that may be used to store a data value and has a name. Variable names are names given to different memory locations that store different kind of data. All C variables must be declared in the program before their use. The value associated with a variable may vary during program execution.

For example,

Let 2 be stored in a memory location and a name **sum** is given to it. On assigning a new value 7 to the same memory location **sum,** its earlier contents are overwritten, since a memory location can store only one value at a time. Figure 4 illustrates this :

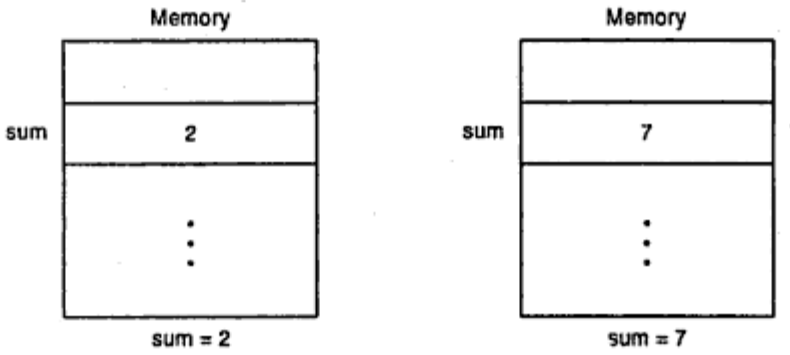


Fig. 4. *Illustrating change in value of a variable.*

Remember, in C the name of a variable has nothing to do with its type. Variable names should be user friendly.

For example,

sum
num
count
average
principal
rate
time

If we want to calculate the sum of two numbers, the variable can be named as 'sum' rather than 'tangura' or some other cryptic (difficult to understand) name.

Rules for Defining Variables

As seen earlier, the rules for constructing different types of constants are different but for constructing variable names in C the rules are same for all types. The following rules must be followed while naming variables :

(*a*) A variable name consists of alphabets, digits and the underscore (–) character. The length of variable should be kept up to 8 characters though your system may allow up to 40 characters.

(*b*) They must begin with an alphabet. Some systems also recognize an underscore as the first character.

(*c*) White space and commas are not allowed.

(*d*) Any reserved word (keyword) cannot be used as a variable name.

For example,

simple_interest, avg_marks, total_salary are valid but these should be written as si_int, av_marks, tot_sal for sure recognization by the compiler.

Declaration of Variables

As stated earlier, all variables in a C program **must** be declared before their use. The type of the variables must be specified at the beginning of the program. The variables are called symbolic variables because these are named locations. The declaration of a variable in C informs about :

1. The name of the variable
2. The type of data to be stored by the variable.

Following are the examples of type declaration statements :

```
int    i, j;
float  x, y;
char   choice;
```

There are two values associated with a symbolic variable :

1. Its data value, stored at some location in memory. It is sometimes referred to as a variable's **rvalue** (pronounced "are-value").
2. Its location value; *i.e.*, the address in memory where the data is stored. It is sometimes referred to as a variable's **lvalue** (pronounced "el-value").

Figure 5 illustrates the concept of *rvalue* and *lvalue*. (The variable *x* has been stored at location number 2062 and variable *j* at location number 2065).

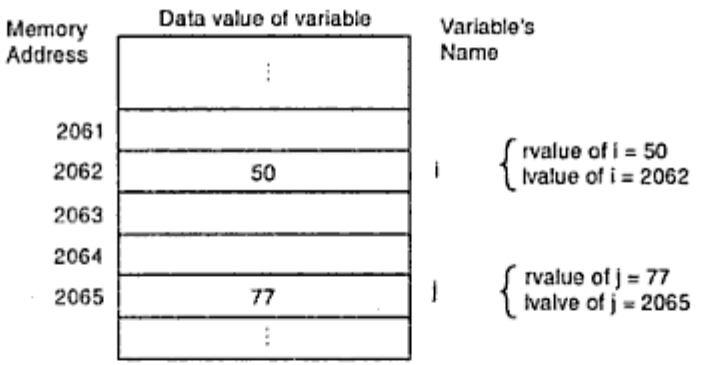


Fig. 5. *The rvalue and lvalue of a variable*

An *lvalue* is an expression to which you can assign a value. The left of an = (assignment operator) must be an lvalue.

> ***Note :*** *Remember that constant values and constant identifiers (declared using const keyword) are not lvalues and can only appear to the right side of an assignment operator.*

A meaningful name given to a variable always helps in better understanding of the program. A variable can be used to store a value of any valid data type. The general syntax for declaration of a variable is given below :

```
type var1, var2, ...., varn;
```

Here, type specifies the data type such as int, float, char etc., and var1, var2, ..., varn are variable names separated by commas. Note that the declaration statement must end with a semicolon.

Structure of a C Program

A **C program** is a collection of functions. A function is a collection of statements tied together to perform a specific task. A user can create the function(s) first and then put them together to code any C program. An overview of the structure of a C program is given below :

Documentation
Header files
Symbolic constants
Global variables

`return-type main(list of arguments)`
`{`
`  Declarations`
`  Executable statement(s)`
`}`
`Function subprograms (user defined)`
`return-type func1(list of arguments)`
`{`
`  body of function`
`}`
`return-type func2(list of arguments)`
`{`
`  body of function`
`}`

A program in C may not contain all the sections shown above but main( ) function is a must as the program execution always starts with main( ). Although *main( )* is not a keyword, treat it as if it were. For example, do not try to use **main** as the name of a variable because you will probably confuse the compiler.

The documentation part contains one or more comments specifying the objective of the program, author name and date etc.

Header files are used for inclusion of functions from the C language library.

Symbolic constants are used for better understanding of certain names that are not going to change throughout the program.

Global variables can be used by all the functions.

The main( ) function has control over the entire program. It must have at least one statement in the executable statement(s) part and the statement(s) in main( ) are enclosed between braces ({ and }). All the statements end with a semicolon (;). If no return-type is specified before the function *main( )*, it returns an integer value to the operating system.

All the function subprograms are user defined with an optional list of *arguments* enclosed in parentheses and generally placed after the main( ) definition is over. These can be placed in any order.

The syntax diagram shown in Figure 6 illustrates the structure of a C program :

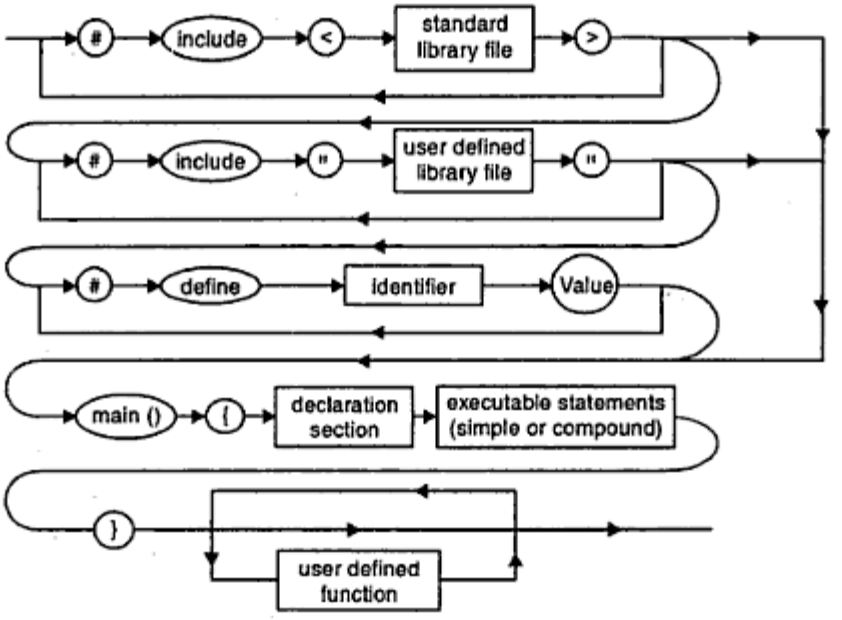


Fig. 6. *Syntax diagram of a C program.*

The structure of a C program can be easily understood with the help of the above syntax diagram. The comments may appear anywhere in a C program whenever necessary. Avoid excessive comments also. The structure of a C program will be more clear with the help of an actual program. The program can be executed manually or on a computer for the deep insight into the various sections of it.

The following C program illustrates some of the concepts explained so far :

Program 1

```
/* swap (interchange) two integer numbers */
/* program written by J.B.Dixit on January 1,2009 */

#include<stdio.h> /* header file included */
void main( ) /* main( ) function definition */
{
   int a=10,b=20,temp; /* variables declared */
   clrscr( ); /* library function to clear screen */
   printf("Numbers before swapping are\n");
   printf("\n%d %d\n",a,b);
   /* swapping */
   temp=a;
   a=b;
   b=temp;
```

```
    printf("\nNumbers after swapping are\n");
    printf("\n%d %d\n",a,b);
    getch( ); /* freeze the monitor */
}
```

Output

```
Numbers before swapping are
10  20
Numbers after swapping are
20  10
```

You may not get some of the concepts at the first glance. Do not panic, you will get a clear idea of writing user friendly programs in C after going through this book.

Comment about the program should be enclosed within /* */. For example, the first two statements in program are comments. Comments are not necessary but it is a good practice to begin a program with a comment, indicating the purpose of the program. Any number of comments can be written at any place in a C program but always remember that **comments cannot be nested.** We can split a comment over more than a line. Such a comment is often known as multi-line comment.

The **#include<stdio.h>** directive is a preprocessor directive is used to include the header file **stdio.h** into the program, which contains functions that perform input-output operations. The # sign is called hash and a statement that begins with this sign is known as the preprocessor directive.

main( ) is a collective name given to a set of statements. It is a special function and must be present in every C program. Whenever a C program is run, it is the first function to be executed. *void* means the function does not return any value to the operating system.

C is a free-form language i.e., the statements of a C program can be typed anywhere in a program file but in precise order. The variables have been declared before using them (a, b and temp). Every C statement must end with a ; (statement terminator). More than one statements are also allowed on a single line. But you will accept that the bad programming style makes a program difficult to understand and debug (remove errors).

Always write the C program in lowercase letters except the symbolic constants. The program should be well indented (horizontally spaced) with braces and statements for easy readability and debugging.

Library function **printf( )** is used to display the output on the screen. The syntax of printf( ) function is given below :

```
printf("format specification string", list of variables);
```

where list of variables is comma separated (even constants are also allowed) and the format specification string contains % followed by a conversion character. For example, %d is used for printing integer values.

The newline character ('\n') takes the cursor to the next line. It is quite useful when we wish to format the output properly on separate lines.

Programming Environment

The Programming environment is the environment used for creating, compiling, linking and executing or running of programs. For example, Unix, Linux and Windows environment. Turbo C and Borland C provide an IDE (*Integrated Development Environment*) for developing and editing a program. On the DOS system **edlin** or any other editor present or a word processor system in non-document mode can be used. On UNIX, you can use **vi** or **ed** text editor for creating and editing the source code. *You must consult the operating system manual which you are having on your system.*

There should be a proper file name extension for a C program. Turbo C and Borland C use extension **.c** for a C program. For writing and executing programs in Turbo C, the Turbo C software must be stored on the hard disk of your computer. Note that the language C++ is a superset of C language and a C program can also be compiled and run using a C++ compiler. Also note that ANSI (American National Standards Institute) has published the C language as a standard and is called as ANSI C. All statements and functions given in ANSI C are accepted in these compilers. After booting your computer there are two ways of getting to the IDE of Turbo C. Obtain the DOS prompt on the screen. Change to the required directory (say TC on the C:/> drive of your hard disk) as shown below :

C:\TC>

Now type TC and press Enter key *i.e.*, C:\TC>TC ↵

It will take you to the IDE of Turbo C.

Or

The procedure of running C program in Windows environment is discussed later on in this chapter.

The steps to run C program in UNIX environment are given below :

All the UNIX systems have a C compiler called cc. It is a command line compiler and is used to generate the executable file.

(*i*) Type **$vi try.c** to invoke the editor. (Here **try.c** is the file name of the program.)

(*ii*) Press **Esc + i** to insert the program.

Executing a C Program

A program coded in C language is executed with the help of following steps :

(*i*) Create the source code.

(*ii*) Compile it.

(*iii*) Link the program with functions using the C library.

(*iv*) Run/execute the program.

Figure 7 illustrates the above mentioned steps with the help of flowchart :

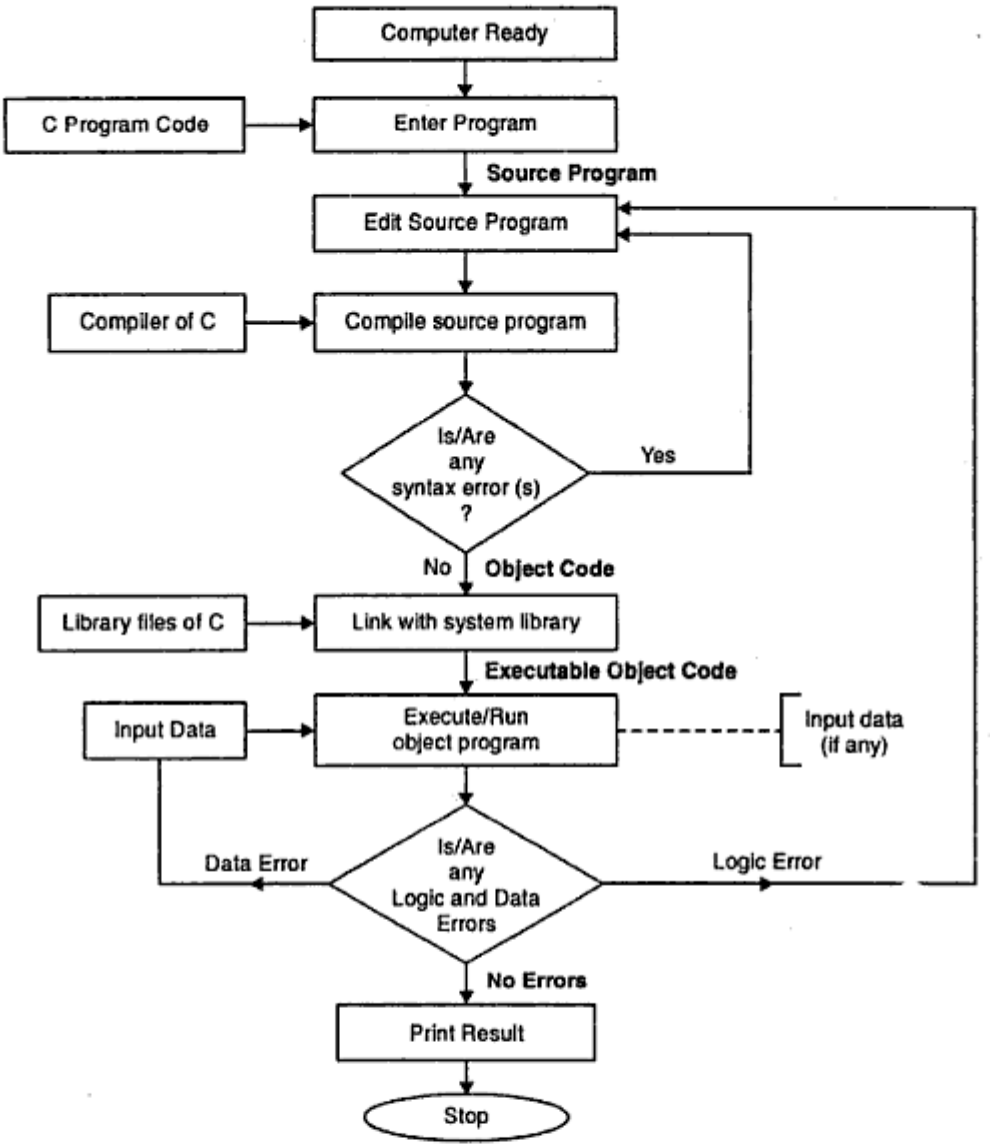


Fig. 7. *Illustration of various steps in compiling and executing a C program.*

Let us explain the above concepts using TURBO C.

Creating Source File

Turbo C provides an IDE (Integrated Development Environment) for developing and editing a program. On the DOS system edlin or any other editor present or a word processor system in non document mode can be used. On UNIX, you can use *vi* or *ed* text editor for creating and editing the C source code.

There should be a proper filename extension for a C program. Turbo C use extension .C for a C program. You must consult the operating system manual which you are having.

Figure 8 shows the IDE screen of TURBO C

The following steps should be followed for creating a source file :

1. You must have developed the program (source code) that you want to store in the file.
2. A suitable filename (of your choice) should be selected for storing the program.
3. Enter the program in the computer and save it in the file you have selected, known as source code file.

Select file option of the main menu and then select the submenu option F3 for loading the desired file into the editor.

Suppose the selected filename is **FUN.C** for the source file. Then after creating it and entering the program the editor screen looks as shown in Figure 9.

Fig. 9. *Turbo C IDE showing a program.*

The preprocessor program processes the source program file (FUN.C in our example) before passing it to the compiler for compilation. In most modern compilers the preprocessor program is integrated into the compiler program.

Compiling and Linking

It depends upon the operating system being used. In Turbo C, we compile the program under the option Compile, as shown in Figure 10. On pressing the Enter key on Compile option, we get :

Fig. 11. *Turbo C IDE after Compilation.*

The following steps are followed while compiling the source code file :

1. The source code is compiled and the translated code (the compiler does it) is known as **object code.** If errors are there, debug them and compile again.

   Any program having syntax (grammatical) errors cannot be compiled successfully.

2. The object code is linked with other library code which are needed for execution of the program. The resulting code is known as **executable code.** If some error(s) occur during linking, debug them and compile the program again.

We will get a file FUN.OBJ after successful compilation and after the linking FUN.OBJ will be linked and a file named as FUN.EXE will be created (which is an executable file).

Compilation

programs in C are completely contained within one source file. However, as length grows, so does its compile time. Hence, C allows a program to be stored in files and lets you compile each file separately. After compilation of all files, they are along with any routines, to form the complete object code. The advantage of compilation is if you change the code in one file, you do not need to recompile program. but the most simple projects, it saves a lot of time. The user to your will contain instructions for compiling multiple-file programs.

the

select the **Run** for getting the result as shown in Figure 12.

12. *Turbo C IDE showing Run Submenu.*

<Ctrl + running the program. After execution of the program the output the user <Alt + F5> to see the user screen, which displays the following :

to the of C programming

Now press any return to editor screen. To exit from it press <Alt + X>.

The following steps are followed while executing the executable file :

1. Execute/Run the exe file and result is obtained (if no error(s) present).
2. The program is debugged in case there is some error.
3. In case of errors the compilation step is repeated again and then the execution step.

Types of Errors and Debugging

Errors may be made during program creation even by experienced programmers. Such type of errors are detected by the compiler. Debugging means removing the errors. The errors are categorized in four types :

(*i*) Syntax errors

(*ii*) Linking errors

(*iii*) Execution time errors (Run time errors)

(*iv*) Logical errors.

(*i*) **Syntax Errors.** These are detected by the compiler. If any grammatical error is made in the program, then during compilation step the compiler will display an error message.

For example,

```
/* illustration of C sample program */
#include<stdio.h>
main( )
{
   printf("Welcome to the world of C programming")
}
```

The above program will not compile successfully as ; is missing after the statement in the main( ) function. So first put a ; after the statement and then compile and run it again for getting the output on the screen.

(*ii*) **Linking Errors.** These may occur during the linking process. For example, if we call a function in main( ) which is not defined then a linking error will be displayed. Correct it and then go ahead.

(*iii*) **Execution Time Errors.** Successfully compiling and linking a program do not mean that the desired result will be displayed. The results may be wrong due to error(s) in logic or division by zero, square root of a negative number is calculated which is not feasible on the system etc. You should check out for such errors from your teacher or consult the user manual for debugging.

(*iv*) **Logical Errors.** These may occur due to error(s) made by the programmer in the logic of the program. The results are wrong if such error(s) is/are present in the program. For example, if the number of days in a month are entered as 32 and no check is made for the days in a month (which are used for calculating wages of an employee), then the calculation is certainly wrong. So, it is the duty of the programmer to avoid such errors for getting accurate result.

> ***Note :*** *Logical errors are most difficult to debug.*

Data Types in C

A data type is a finite set of values along with a set of rules for allowed operations.

C supports several different types of data, each of which is stored differently in the computer's memory. Data types in C are shown with the help of figure 13 :

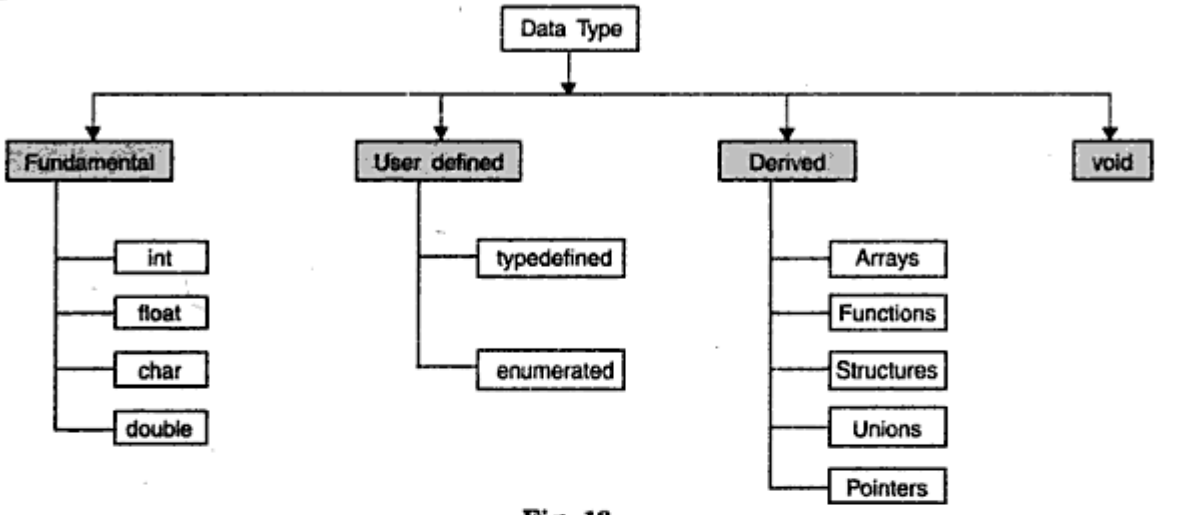


Fig. 13.

The fundamental and user defined data types have been discussed below. The remaining data types will be discussed in the due course.

The Four Fundamental Data Types

Fundamental data types include the data types at the lowest level *i.e.*, those which are used for actual data representation in the memory of the computer. All other data types are based on the fundamental data types.

The fundamental data types in C are :

char — for characters and strings
int — for integers
float — for numbers with decimals
double — for double-precision floating numbers.

Since, the above data types are fundamental *i.e.*, at the machine level, the storage requirement is hardware dependent. Table 6 shows the storage requirement of the above data types on a 16-bit machine :

Table 6. Storage requirement of fundamental data types

Data type	*Size (bits)*	*Range of values*
char	8 (1 byte)	– 128 to 127
int	16 (2 bytes)	– 32768 to 32767
float	32 (4 bytes)	– 3·4e38 to 3·4e38
double	64 (8 bytes)	– 1·7e308 to 1·7e308

Table 7 represents the various data types in C-basic data types and qualified. A qualifier changes the characteristics of the data type, such as its size or sign.

Table 7. Data Types in C

Keyword	*Representation of data type*
char	a single character
int	an integer
float	a single precision floating point number
double	a double precision floating point number
unsigned char	an unsigned single character
signed char	a signed single character
signed int	a signed integer
unsigned int	an unsigned integer
long int	a long integer
short int	a short integer
long double	an extended precision floating point number
signed short int	a signed short integer
unsigned short int	an unsigned short integer
signed long int	a signed long integer
unsigned long int	an unsigned long integer

For altering the size we use — **short** and **long**

For altering the sign we use — **signed** and **unsigned**

Generally size qualifiers cannot be used with data types **float** and **char,** and sign qualifiers are not applicable with **float, double** and **long double.** Also note that the precision means the number of significant digits after the decimal point.

Here **signed** means that the variable can take both + and – values and **unsigned** means only + values are allowed. The keywords **short** means the variable takes lesser number of bits and **long** means greater number of bits.

Remember that **char** or **signed char** mean the same, **int** or **short** or **short int** or **signed short int** mean the same, **long** or **long int** or **signed long int** mean the same. If we use **short, long** or **unsigned** without a basic data type specifier, the data type is taken as an **int** type by the C compilers.

The following program segment illustrates the declaration of different types of variables :

```
void main( )
{
   int i,j;
   float  a,b;
   char  grade;
   short int  n;
   long int  population;
   double   value ;
   - - - - - - /* executable statements */
}
```

Note : All floating arithmetic computations in C are carried out in double mode. Whenever, a ***float*** *appears in an expression, it is changed to double mode. When a double has to be converted to float, double is rounded before truncation to float length.*

User-Defined Type Declaration

In C, we can define an identifier for representing an existing data type. It is known as "type definition" and can be used for declaration of variables. The syntax for type definition is given below :

typedef type identifier;

Here, type represents an existing data type and identifier represents the new name in place of type. Remember that **typedef** is not capable of creating any new data type but provides only an alternative name to an existing data type. *For example,*

```
typedef    int    number;
typedef    float  amount;
```

The above statements tell the compiler to recognize **number** as an alternative to **int** and **amount** to **float.**

Now, the variables can be created using **number** and **amount** as given below :

```
number  num1, num2;
amount  fund, loan, instalment;
```

The significance of **typedef** is that we can provide suitable names to existing data types and make the code easier to read and understand.

Using **typedef** does not replace the existing standard C data type name with the new name, rather we can now use both for creating variables.

Enumerated Data Types

C provides another user-defined data type known as "enumerated data type". It attaches names to numbers, thereby increases the readability of the program. It is generally used when we know in advance the finite set of values that a data type can take on. The syntax for enumerated data type is given below :

enum identifier {val1, val2, ..., valn};

Here, identifier represents the user defined enumerated data type and val1, val2, ..., valn are called members or enumerators.

Now, we can declare variables using the above declared type :

enum identifier var1, var2, ..., varn;

The variables var1, var2, ..., varn can take only one value out of val1, val2, ..., valn.

For example,

```
enum  months  {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
enum  months  month1, month2; /* variables declared of type months */
```

Enumerated means that all values are listed. The enumerators are automatically assigned values starting from 0 to n–1. Thus the first value Jan. will have 0, the second value Feb. will have value 1, and so on, lastly the value Dec will have value 11.

The assignment, arithmetic and comparison operations are allowed on enumerated type variables. *For example,*

```
month1 = Jan;
month2 = Jul;
printf ("%d", month2-month1);
```

The above output statement will print 6 as Jul. has value 6 and Jan. a 0 with it.

The following statement will work as well,

```
if (month1 < month2)
        ----------
else
        ----------
```

We can change the default value assignment by assigning the integer values to the enumerators. *For example,*

```
enum months {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

Now the enumerators will have values from 1 to 12 respectively.

The default ordinal values can be changed for more than one enumerator also.

We can combine the definition and declaration of enumerated variables.

For example,

```
enum months {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}
                                                        month1, month2;
```

The values assigned to the enumerators need not be distinct, increasing or positive. *For example,*

```
enum values {mid = 16383, low = 0, high = 32767};
```

> ***Note :*** *The input and output of variables of enumerated data type is not allowed in C.*

Derived data types will be discussed later on.

Storage Class Declaration

The storage class in C provides the complete information about the location and visibility of variables. Scope of a variable means the portion of the program within which it can be referenced and lifetime means the time of its existence in the memory.

Consider the following program segment :

```
int x;  /* global variable */
void main( )
{
   int i,j;  /* local variables */
   float a;
   ----------
   ----------
   func1( )
}
```

```
func1( )
{
    int i,j; /* local variables */
    float amount;
    - - - - - - - - - - -
    - - - - - - - - - - -
}
```

A variable declared outside the main( ) function can be used by all the functions without declaring it and is known as a **global** or *external* variable. A variable declared inside a function can be used only inside it and is known as a local variable. Two different functions can have the same variable names but these are confined to their own functions only. The four storage classes in C are given in Table 8.

Table 8. Storage Classes in C

Storage class	*Purpose*
auto	Variable used as a local variable. This is the default one. Initial value of variable is garbage value without initialization.
static	Variable used as a local variable. Retains its value during next function call. Default initial value is 0.
register	Variable used as a local variable. May be stored in register if possible. Default initial value is garbage value.
extern	Variable used as a global variable. Default initial value is 0.

The storage classes can be used with variables as given below :

```
auto      int     i, j;
register  char    choice;
extern    float   sum;
static    long    n;
```

Assignment Operator

An operator is a symbol that operates on a certain data type.

In C, the '=' symbol is known as the assignment operator. It sets the value of the variable on the left hand side of it to that of the right hand side of it.

Assignment Expressions and Assignment Statements

In C, an expression is a combination of variables, constants and operators written according to the syntax. An expression always results in some value of a certain type that can be assigned to a variable using = operator.

For example,

```
interest = (prin * rate * time)/100.0;
```

Here, the variables prin, rate, time used in the expression on the right hand side of = must be declared with data types and assigned values before the execution of the above statement. The above statement is an example of an assignment statement.

C supports a variety of assignment statements. These are given below :

(*a*) Simple assignment statement

(*b*) Multiple assignment statement

(*c*) Arithmetic assignment statement.

Simple Assignment Statement

The syntax of simple assignment statement is

```
var_name = expression;
```

C treats = operator like other operators.

For example,

```
int  i, j, s;
i = 20;
j = 30;
printf ("\nValue of s is %d\n", s = i + j);
```

The above statement is valid and prints the result as :

```
Value of s is 50
```

First the values of i and j are added, assigned to s and then the value of s is printed.

Remember that the type of var_name and expression must be compatible.

Let us consider the following variable declarations :

```
int i, j ;
float x, y, avg;
```

Now the following assignment statements are valid :

```
i = 55;
j = 250;
x = 14.0;
y = 30.0;
avg = (x + y)/2.0;
```

The following are invalid assignment statements :

```
x + y = avg; /* x + y is not a valid variable name */
avg = 'A'; /* avg is not a character variable */
```

In C, a variable can be initialized when declared. The syntax of this is :

```
type var_name = constant;
```

For example,

```
int sum = 0;
float avg = 0.0;
char ans = 'y';
```

Multiple Assignment Statement

In C, the same value can be assigned to more than one variables of same type using multiple assignment statement. The syntax of a multiple assignment statement is given below :

```
var_name1 = var_name2 = var_name3 = constant;
```

Here var_name1, var_name2, var_name3 are variable names and constant is the value to be assigned. It can be extended for more than three variables also. The value is assigned from right to left.

For example,

```
int i, j, k, n;
i = j = k = n = 0;
```

Arithmetic Assignment Statement (Self Replacement Statements)

When the variable itself takes part in the assignment statement and stores the result in itself, the arithmetic assignment statement comes into picture.

For example,

```
int  num1, num2;
num1 = 100;
num2 = 250;
num1 = num1/20;     /* arithmetic assignment statement */
num2 = num2 * 4;    /* arithmetic assignment statement */
```

The following program illustrates the various concepts described so far :

Program 2

```
/* illustration of variable declarations and assignments */

#include<stdio.h>
void main( )
{
   /* variable declarations */
   int i,j,s;
   float x,y,avg;
   char ans,choice;
   clrscr( ); /* clears the screen */
   /* simple assignments */
   x=22.5;
   y=17.5;
   avg=(x+y)/2.0;
   /* multiple assignments */
   i=j=s=500;
   choice=ans='A';
   /* arithmetic assignments */
   i=i*2;
```

```
    j=j-50;
    s=s/100;
    /* print the results */
    printf("%d %d %d\n\n",i,j,s);
    printf("%f %f %f\n\n",x,y,avg);
    printf("%c %c\n",ans,choice);
}
```

Output

```
1000  450  5
22.500000  17.500000  20.000000
A  A
```

The printf( ) function will be discussed in Chapter on Input and Output Functions.

Arithmetic Conversion

In C, the type of value on the right hand side of the = (assignment operator) is converted to that of the type on the left hand side of it. It may involve anyone of the following :

Promotion

Truncation

The conversion of a variable of lower type (which stores lower range of values or has lower precision) to a higher type (which stores higher range of values or has higher precision) is known as promotion.

Truncation means discarding the fractional part when the real value is converted into an integer (higher type to lower type).

For example, consider the following statements :

```
int i;
float x;
i = 20;
x = i;  /* int converted to float type (promotion) */
```

The value of i is assigned to x which is a float type variable. The compiler converts the integer value to float automatically.

Another type of promotion called integral promotion takes place when the character variable is assigned to an integer. *For example,*

```
int i;
char ch;
ch = 'A';
 i  = ch;   /* char converted to int type (integral promotion) */
```

The above statement assigns the ASCII value of 'A' *i.e.,* 65 to the integer type variable i.

Consider the following statements

```
int j;
float y;
```

```
y = 15·6;
j = y;      /* float converted to int type (truncation) */
```

The value of j after the execution of the above statement will be 15. The fractional part *i.e.,* ·6 is discarded.

In all the above cases the conversion takes place automatically. We can forcibly convert one type of data to another in C. *For example,*

```
int a, b;
float result;
a = 77;
b = 12;
result = a/b;
```

After execution of the above statements, result will store 6·0 due to the fact that the right hand side expression involves only integers. First integer division takes place and then assignment. For getting the accurate result, we may convert anyone out of a and b to a float value as shown below :

```
result = (float)a/b;
```

The conversion of one type of data into another type is known as **typecasting.** The general syntax of typecasting is :

```
(type) var_name
```

Here, type enclosed within parentheses specifies the type to which var_name is to be converted.

Typecasting is possible from a higher type to a lower type also. *For example,*

```
float x = 50·8;
(int) x
```

The above expression evaluates to 50 converting a float type to int type.

After typecasting the value of the variable remains unchanged.

Declaring a Variable as Constant

In certain situations we may like that the value of some desired variables must remain constant during the program execution. This can be done by using the qualifier **const** at the initialization time.

For example,

```
const int S = 50;
```

Here, **const** is a data type qualifier defined by ANSI standard.

The value of variable S will remain constant throughout the program.

The value of S can be used in expressions on right hand side of assignment statements as that of a variable.

Similarly the following statements will work :

```
const char ch = 'y';
const float value = 3.14159;
```

Declaring a Variable as Volatile

The qualifier **volatile** defined by ANSI standard declares a variable as volatile. *For example,*

```
volatile int time;
```

Here, volatile refers that the value of the variable time may be altered at any moment by the external sources (from outside the program).

The contents of the variable may be changed by some external sources whenever time appears either on the left or right of an assignment statement. The value of the variable declared as **volatile** is checked everytime before use and the altered value is used.

A **volatile** variable can be changed by the source program also (*i.e.,* the program that contains it). In case we want it to be altered by external sources only, define it as given below :

```
volatile const int target = 555;
```

Symbolic Constants (Constant Identifiers)

Sometimes we require to use a constant value at many places in a C program. In doing so we face two major problems :

(*i*) Modification problem

(*ii*) Understanding problem.

Modification Problem

The first problem is due to the fact that if the value of the constant is to be changed later on then correction is required at as many places as it exists. If it is not changed unknowingly at few places then the program will produce wrong results.

Understanding Problem

With a numeric value it is not always clear, what does it mean ? After sometime even the programmer might forget the actual purpose of its use.

A symbolic name avoid such serious problems. The constant value can be given a symbolic name at the start of the program to avoid any sort of confusion. Now we can use these symbolic names in the program with ease and modification at any later stage is very easy. A symbolic constant is defined in the following way :

```
#define symbolic_name   constant value
```

For example,

```
#define  S   20
#define  PI  3.14159
#define  FLAG  1
```

Here, S, PI, FLAG represent symbolic names or constant identifiers.

The symbolic names do not appear in the declaration section as these are constants and the above definition is sufficient.

The rules with a **#define** statement for defining a symbolic constant are :

(*a*) These are like variable names : Use uppercase letters for better readability of the program though lowercase letters are allowed.

(*b*) Always type #define together without any space in between.

(*c*) The first character must be # in the line of definition.

(*d*) One or more blank(s) are required after #define and symbolic_name.

(*e*) **#define** statement **must not** terminate with a ; (semicolon)

(*f*) No other value should be assigned to it during the program.

(*g*) The type of symbolic name depends on the type of the constant value associated with it.

(*h*) **#define** statement should be placed at the start of the program but we can place them anywhere before the statement using it.

Following are some invalid symbolic constant definitions :

```
#define   SIZE = 10    /* = not allowed */
# define  M 5          /* White space between # and define not allowed */
#Define   N 50         /* Define should be in lowercase */
```

Summary

- C is a general-purpose, structured programming language.
- C is often called a *middle-level* language.
- C was originally developed in 1972 by Dennis Ritchie at Bell Telephone Laboratories, Inc. (now AT&T Bell Laboratories).
- Every C program consists of one or more *functions,* one of which *must* be called main. The program will always begin by executing the main( ) function.
- *Comments* (remarks) may appear anywhere within a program and these are placed within the delimiters /* and */ (*e.g.,* /* it is a comment */).
- Each compound statement (two or more statements together) is enclosed within a pair of braces, *i.e.,* {and}.
- Each expression statement must end with a semicolon (;).
- C language is *case sensitive i.e.,* uppercase and lowercase character are not equivalent.
- C is a free form language.
- Identifiers are names given to various program elements, such as variables, functions and arrays.
- All variables must be declared for their types before using in a C program.
- Do not use keywords or any system library names for identifiers.
- Give meaningful names to identifiers to make the program user friendly and easy to modify.
- All variables must be initialized before using them in a C program.
- Some implementations of C language recognize only the first eight characters, though most implementations recognize more (typically, 31 characters). The ANSI standard recognizes 31 characters.
- C supports several different types of data, each of which may be represented differently within the computer's memory. The basic data types are *int, char, float* and *double.*

- C has four basic types of constants, these are *integer constants, floating-point constants, character constants* and *string constants.*
- An escape sequence always begins with a backward slash (\) and is followed by one or more special characters. For example, **\n** (newline).
- Initial values can be assigned to variables within a type declaration.
- A *statement* causes the computer to perform some action.
- A *symbolic constant* is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant or a string constant.
- Generally symbolic names are written in uppercase, to distinguish them from ordinary C identifiers.
- The *#define* feature, which is used to define symbolic constants, is one of several features included in the C *preprocessor* (*i.e.,* a program that provides the first step in the translation of a C program into machine code).
- Don't use any space between # and **define.**
- Don't use semicolon at the end of **#define** directive.
- A global variable (variable defined before main( )) is available to all the functions in the program (provided it is not declared in other function(s)).
- A local variable (variable defined within a function) is available to the function within which it is defined but not to other functions.
- Logical errors are most difficult to debug.

Review Questions and Exercises

1. Why 'C' language is named so ? What is the basic structure of a C program ? Also discuss the importance of 'C' language over other languages.
2. Give some advantages and disadvantages of C language.
3. What does the C character set consists of ?
4. What is C token ? List the different types of C tokens.
5. What are the identifiers in any programming language ? How the identifiers can be formed in C ?
6. How error handling is performed in C ? What is its scope and limitations ?
7. What are various data types used in C language ? Illustrate their declaration and usage with appropriate examples.
8. Illustrate the significance of keywords and identifiers in C.
9. (*a*) Name and describe any four basic data types in C.
   (*b*) Name and describe the four data type qualifiers. To which data types can each qualifier be applied ?
   (*c*) What are constants ? Explain any four basic types of constants.
10. Describe the basic data types supported by 'C' language. How many characters can be included in an identifier name ? Are all of these characters equally significant ?

11. (*a*) What are different types of integer constants ? What are long integer constants ? How do these constants differ from ordinary integer constants ? How can they be written and identified ?

(*b*) Describe two different ways that floating-point constants can be written in C. What special rules apply in each case ?

(*c*) What is a character constant ? How do character constants differ from numeric type constants ? Do character constants represent numerical values ?

12. (*a*) What is a string constant ? How do string constants differ from character constants ? Do string constants represent numerical values ?

(*b*) What is meant by backslash characters ? What is there utility in programs ?

(*c*) What is a variable ? How can variables be characterized ? Give the rules for variable declaration.

(*d*) What is the purpose of type declarations ? What are the components of type declaration ?

13. What are enumerated data type ? What are there applications ? Give examples.

14. What is meant by enumerated data type ? Explain the concept with suitable example.

15. What is type casting ? Discuss with example.

16. What are basic storage classes for C variables ? Explain briefly.

17. What is a symbolic constant ? How is a symbolic constant defined ? How is the definition written ? Where must a symbolic constant definition be placed within a C program ?

18. Explain briefly the salient features of a structured language. Which programming method is followed in C language ?

19. State the meanings of the following keywords in C :

auto, double, int, long.

20. Give an example for enumerated data type.

21. (*a*) State whether the following statements are true or false. If a statement is false, then write the correct version :

(*i*) main( ) is where the program begins its execution.

(*ii*) A line in a program may have more than one statement.

(*iii*) A printf statement can generate only one line of output.

(*b*) What is a symbolic constant ? Explain its use, supporting your answer with examples.

❑❑❑

Operators and Expressions

Introduction

Operators are the verbs of a language that help the user perform computations on values. C language supports a rich set of operators. Different types of operators discussed in this chapter are :

1. Arithmetic operators
2. Unary operators
3. Relational operators
4. Logical operators
5. Assignment operators
6. The conditional operator
7. Bitwise operators

An expression is a formula consisting of one or more operands and zero or more operators connected together to compute the result. An operand in C may be a variable, a constant or a function reference. For example, in the following expression,

x – y

minus character '–' is an operator and x, y are operands.

Arithmetic Operators

These are used to perform arithmetic operations. All of these can be used as binary operators. These are :

+	add	
–	subtract	
*	multiply	
/	divide	(the divisor must be non zero)
%	modulo	(gives remainder after division on integers)

The parenthesis( ) are used to clarify complex operations. The operators + and – can be used as unary plus and unary minus arithmetic operators also. The unary – negates the sign of its operand.

> ***Note :*** *C language has no operator for exponentiation.*

The function **pow** (x, y) which exists in **math.h** returns x^y.

Following are some examples of arithmetic operators :

x + y, x – y, x * y, x/y, x % y, – x * y

Here x and y are operands. The % operator cannot be used on floating point data type.

Arithmetic Expressions

An expression consisting of numerical values (either any number, variable or even some function call) joined together by arithmetic operators is known as an arithmetic expression. For example, consider the following expression :

(x – y) * (x + y)/5

Here x, y and 5 are **operands** and the symbols –, *, +, / are **operators.** The precedence of operators for the expression evaluation has been given by using parentheses which will over rule the operators precedence. If x = 25 and y = 15, then the value of this expression will be 80.

Consider the following arithmetic expression :

3 * ((i%4) * (5 + (j – 2)/(k + 3)))

where i, j and k are integer variables. If i, j and k have values 9, 14 and 6 respectively, then the above expression would be evaluated as

3 * ((9%4) * (5 + (14 – 2)/(6 + 3)))
= 3 * (1 * (5 + (12/9)))
= 3 * (1 * (5 + 1))
= 3 * (1 * 6)
= 3 * 6
= 18

Instead of writing such lengthy expressions it is better to break it into several shorter expressions for easy coding.

Modes of Expression

In C, we can have an arithmetic statement of the following types :

(*i*) Integer mode arithmetic statement
(*ii*) Real mode arithmetic statement
(*iii*) Mixed mode arithmetic statement

Integer Mode Arithmetic Statement

It consists of all operands as either integer constants or integer variables. *For example,*

```
int x, y, i, j, result;
x = 50;
y = x + 10;
i = 2;
```

```
j = i * 3;
result = x/7 - y%j + i;
```

In case of integer division, the result is truncated towards zero when both the operands are of the same sign. When one of the operands is negative the truncation depends upon the implementation. *For example,*

3/5 = 0 and – 3/– 5 = 0

but – 3/5 can result into 0 or – 1 (it is machine dependent).

For modulo division, the sign of result is always that of the first operand or dividend. *For example,*

```
 13% – 5 =  3
–13 % – 5 = – 3
 – 13% 5 = – 3
```

Real Mode Arithmetic Statement

It consists of all operands as either real variables or real constants.

For example,

```
float  a, b, d;
a = 5.0;
b = a * 20.5;
d = a/2.0 * b – 10.0;
```

In C, a real operand may store value in either of the two forms *i.e.,* in fractional form or exponential form. The rounding takes place in such cases and the final result of an expression in real mode is always an approximation of the calculation performed.

> ***Note :*** *We cannot use % operator with real operands.*

Mixed Mode Arithmetic Statement

It consists of operands of both types *i.e.,* integers and reals. When anyone of the two operands is real, the expression is known as a mixed mode arithmetic expression and the operation performed in such a case results in a real value as the answer. For example, consider the following division :

24/10.0 = 2.4

If we write 24/10, then the result will be 2 as seen earlier.

The following points must be taken into account when we write arithmetic instructions or expressions :

(*i*) Only one variable is allowed when we use an = (assignment) operator in an arithmetic statement. *For example,*

```
int  i, j, k;
i = 2;
j = 7;
k = i + j;    /* valid C statement */
i + j = k;    /* invalid C statement */
```

(*ii*) Character constants can also be assigned to character variables like the assignment of integer and float type expressions to **int** and **float** type variables. *For example :*

```
char  ch1, ch2;
ch1 = 'A';
ch2 = 'y';
```

(*iii*) Arithmetic operations are allowed on integer, float and character type values *i.e.*, **int, float** and **char** data types.

(*iv*) Every operator must be explicitly written in formation of an expression. Do not assume it. *For example,*

```
int i, j, prod;
prod = i * j,     /* valid C statement */
prod = i j;       /* invalid statement as * not typed */
```

Arithmetic Operators Precedence (Including Parenthesis)

In C, the arithmetic operators have the priority as shown below :

First priority	* / %	} Arithmetic operators
Second priority	+ –	
Third priority	=	} Assignment operator

The sequence of operations in evaluating an arithmetic expression is also known as hierarchy of operations. This is necessary to avoid any doubt while evaluating an expression. The following precedence rules are followed in expression evaluation :

(*i*) All the subexpressions within the parentheses are evaluated first. Nested parenthesized subexpressions are evaluated inside-out, with the innermost expression being first to be evaluated.

(*ii*) Operators in the same subexpression are evaluated as given below :

*, / and % performed first

+ and – performed next

Any function referenced (*i.e.*, invoked) in the expression gets the highest precedence over all the arithmetic operators.

(*iii*) Operators in the same expression with the same priority are evaluated from left to right.

For example,

Consider the following expression for checking the operators precedence.

```
    15 * 7/(2 – 3*5/7 + 4) – 7 * 9 % 4
  = 15 * 7/(2 – 15/7 + 4) – 7 * 9 % 4
  = 15 * 7/(2 – 2 + 4) – 7 * 9 % 4
  = 15 * 7/4 – 7 * 9 % 4
  = 105/4 – 63 % 4
  = 26 – 3
  = 23
```

The parentheses can change the order of evaluation of an expression. Proper use of parentheses makes a C program expression easy to understand. So whenever in doubt use a pair of parentheses to make the concept clear as there is no penalty on their use.

Unary Operators

C has a class of operators that act upon a single operand to give a new value. These type of operators are known as unary operators. Unary operators generally precede their single operands, though some unary operands are written after their operands.

Unary Minus

Perhaps the most common unary operation is *unary minus,* where a minus sign precedes a numerical constant, a variable or an expression. (Some programming languages permit a minus sign to be included as a part of a numeric constant). In C, however, all numeric constants are positive. Thus, a negative number is actually an expression, having the unary minus operator, followed by a positive numeric constant. Note that the unary minus operation is totally different from the arithmetic operator which means subtraction (–). The subtraction operator requires two separate operands.

For example,

```
– 537
– (a + b)
– 5 * (x + y)
– disc
– 0.5
– 3E – 7
```

Increment and Decrement Operators

Two other commonly used unary operators are the *increment operator,* **++,** and the *decrement operator,* **––,** that operate on integer data only.

The increment (++) operator increments the operand by 1, while the decrement operator (––) decrements the operand by 1.

For example,

```
int i, j;
i = 10;
j = i + +;
printf ("%8d%8d", i, j);
```

Here, the output would be 11 10. First the value of **i** is assigned to **j** and then **i** is incremented by 1 *i.e.,* post-increment takes place. This is shown in Figure 1.

Postfix :

```
j = i++;
```

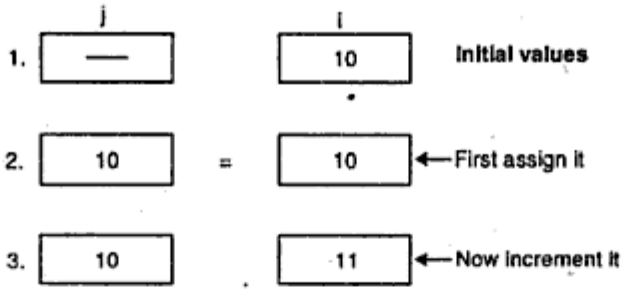


Fig. 1. *The increment operator (postfix form).*

If we have

```
int i, j;
i = 20;
j = + + i;
printf ("%8d%8d", i, j);
```

The output would be 21 21. First the value of **i** is incremented by 1 and then assignment takes place *i.e.,* pre-increment of **i**. This is shown in Figure 2.

Prefix :

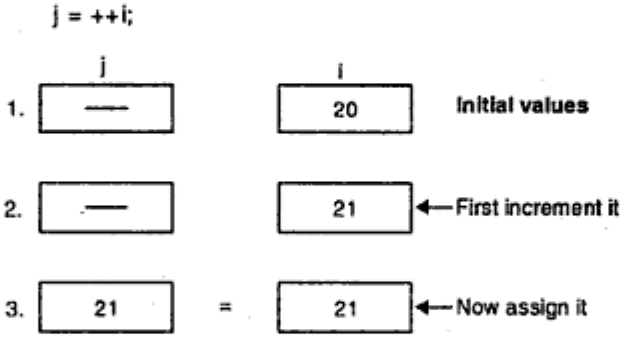


Fig. 2. *The increment operator (prefix form).*

All the three statements given below are equivalent :

```
i = i + 1;
i ++;
++ i;
```

Now let us consider the example for (--) operator :

```
int i, j
i = 10;
j = i --;
printf ("%8d%8d", i, j);
```

Here the output would be 9 10. First the value of **i** is assigned to **j** and then **i** is decremented by 1 *i.e.,* post-decrement takes place.

If we have

```
int i, j;
i = 20;
```

```
j = - - i;
printf ("%8d%8d", i, j);
```

The output would be 19 19. First the value of **i** is decremented by 1 and then assignment takes place *i.e.,* pre-decrement of **i.**

All the three statements given below are equivalent :

```
i = i - 1;
i --;
-- i;
```

Most C compilers produce very fast, efficient object code for increment and decrement operations—code that is better than that generated by using the equivalent assignment statement. Therefore, you should use the increment and decrement operators when you can.

> ***Note :*** *On some compilers a space is required on both sides of + + i or i + +, i- - or- - i. The prefix + + or - - operators follow change - then - use rule and the postfix + + and - - operators follow use-then-change rule.*

Example. *Evaluate y = x++ + ++x if x is 20 initially.*

Solution. y = (x ++) + (++ x) = 21 + 21

In Turbo C, firstly all prefix operators are evaluated prior to expression evaluation. The resultant value of prefix operator is used in the expression and expression is evaluated. Here, ++x evaluates to 21 making the value of **x** as 21. This value of **x** is used at all places in the expression *i.e.,* 21 + 21. After evaluation of the expression, the postfix operator performs its function *i.e.,* after the evaluation of the expression, x++ will make the value of **x** as 22 (21 to 22) = 42. Therefore, y = 42.

The sizeof Operator

It is an unary operator which provides the size, in bytes, of the given operand. The syntax of **sizeof** operator is :

sizeof (operand)

Here the operand is a built in or user defined data type or variable.

The **sizeof** operator always precedes its operand. *For example,*

```
sizeof (float)
```

returns the value 4.

This information is quite useful when we execute our program on a different computer or a new version of C is used. The **sizeof** operator helps in case of dynamic memory allocation for calculating the number of bytes used by some user defined data type.

A *cast* is also considered to be a unary operator. It has been discussed earlier. In general terms, a reference to the cast operator is written as given below :

(type)

Thus, the unary operators we have discussed so far in this book are –, + +, – –, sizeof and *(type).*

Unary operators have a higher precedence than arithmetic operators. Hence, if a unary minus operator acts upon an arithmetic expression that contains one or more

arithmetic operators, the unary minus operation will be carried out first (if parentheses do not enclose the arithmetic expression). Also the **associativity** of the unary operators is right-to-left, though consecutive unary operators rarely appear in simple programs. *For example,*

```
int x, y;
x = 5;
y = 25;
```

The value of the expression – x + y will be – 5 + 25 = 20. Here the unary minus is carried out before the addition operation. If the expression is – (x + y), then it becomes – (5 + 25). The value of this expression is – (5 + 25) = – 30. In this case, the addition is performed first and then the unary minus operation. C has several other unary operators. These will be discussed, as the need arises.

Relational Operators

These are used to compare two variables or constants. C has the following relational operators :

Operator	Meaning
= =	Equals
! =	Not equals
<	Less than
>	Greater than
< =	Less than or equals
> =	Greater than or equals

Logical Operators

In C, we can have simple conditions (single) or compound conditions (two or more). *C considers 0 as false and any non-zero value as a true value. Values 0 and 1 (i.e., false and true) are the truth values of expressions.* The logical operators are used to combine conditions. The notations for these operators is given below :

Operator	Notation in C
NOT	!
AND	&&
OR	¦¦

The unary negation operation ! is useful as a test for zero.

> ***Note :*** *The notation for the operator OR is given by two broken lines. These follow the same precedence as in other languages . NOT (!) is evaluated before AND (&&) which is evaluated before OR(¦¦). Parentheses( ) can be used to change this order.*

Relational and Logical Expressions

An expression involving a relational operator is known as a **relational expression.** The resulting expression will be of integer type, since in C, true is represented by 1 and false by 0. *For example,*

```
int a = 2, b = 3, c = 4;
```

Relational expression	Value
$a < b$	1 (true)
$c <= a$	0 (false)
$b == c$	0 (false)
$c >= b$	1 (true)
$a != c$	1 (true)
$a > 1$	1 (true)

When a relational operation is carried out, different types of operands (if present) will be converted appropriately.

> ***Note :*** *Don't use = for testing equality. The operator for testing equality is == (two = signs together).*

In C, the arithmetic operators have higher priority over relational operators.

Relational operators are used with **if, while, do while** statements to form relational expressions which help in making useful decisions. These statements are discussed later on.

> ***Note :*** *Avoid equality comparisons on floating-point numbers and comparing signed and unsigned values.*

An expression formed with two or more relational expressions is known as a **logical expression** or **compound relational expression.** *For example,*

```
int day;
day >= 1 && day <= 31
```

The logical expression given above is true, when both the relational expressions are true. If either of these or both are false, it evaluates as false.

The following truth tables gives the result of applying **&&, ||** operators on two operands :

Here 1 denotes non-zero or true and 0 denotes zero or false.

The ! (logical NOT) operator negates the value of any logical expression *i.e.*, true expression becomes false and *vice-versa* as given below :

The following examples illustrate the logical expressions.

```
int a = 10, b = 50;
char ch1 = 'A';
float x = 9.5;
```

Logical expression	*Value*
a + x < = b	1 (true)
a < b && ch1 = = 'A'	1 (true)
b – x < 5.0 ¦ ¦ ch1 ! = 'A'	0 (false)

As stated earlier in the above example, the arithmetic operations are carried out prior to relational and logical operations. The relational operations take place before the logical operations.

In the last of the above examples, first operation carried out is subtraction (*i.e.*, b – x), then the comparison (*i.e.*, b – x < 5.0), after that the equality comparison (*i.e.*, ch1! = 'A'), and finally the logical ¦ ¦ is carried out.

While evaluating complex logical expressions (*i.e.*, logical subexpressions) joined by !, && and ¦ ¦) there is no need to evaluate the entire logical expression if its value can be found from its subexpressions. *For example,*

```
float x;
int i;
x = 0.005;
i = 300;
x > 1.5 && i > 200
```

Here only the first subexpression (*i.e.*, x > 1.5) is evaluated which is false so the second subexpression (*i.e.*, i > 200) will not be evaluated, because the value of the entire logical expression will be false due to presence of && operator. Suppose we have

```
x > 1.5 ¦ ¦ i < 400
```

Now, both the subexpressions are evaluated because the first one (*i.e.*, x > 1.5) evaluates as false and due to presence of || operator the value of the entire expression cannot be decided without evaluating the second subexpression (*i.e.*, i < 400) which in this case evaluates to true. So the entire expression is true, using the values of **x** and **i** as shown in the above case.

Precedence of Relational Operators and Logical Operators

Each operator in C has a precedence of its own. It helps in evaluation of an expression. Higher the precedence of the operator, earlier it operates. The operators having same precedence are evaluated either from left to right or from right to left, depending on the level, known as the **associativity** of the operator.

Table 1 summarizes the precedence of relational and logical operators. The higher the position of an operator in the table, higher is its priority or precedence :

Table 1. Precedence of relational and logical operators

Operator	*Type*	*What it does ?*
!	Logical Not	Logical negation (NOT)
< <= > >=	Relational Relational Relational Relational	Less than Less than or equal to Greater than Greater than or equal to
== !=	Relational Relational (Equality)	Equal to Not equal to
&&	Logical AND	Logical AND in a complex expression
||	Logical OR	Logical OR in a complex expression

Assignment Operators

There are several different assignment operators in C. All of these are used to form *assignment expressions,* which assign the value of an expression to an identifier.

The most commonly used assignment operator is = . Assignment expressions which use this operator are of the following form :

identifier = expression

where *identifier* generally represents a variable and *expression* represents a constant, a variable or a more complex expression.

For example,

```
int i = 10;
float a = 5.0;
```

ABOUT THE BOOK

Logical arrangements of contents, clarity of presentation and illustration through abundance of programming examples in the book aid the student and novice in mastering the C language. A number of unique features make this book different from other existing books in the field of C programming.

- Motivating the unmotivated.
- Over 250 tested program examples provided.
- User friendly to enhance the user's understanding of theoretical and practical concepts.
- Algorithms and flowcharts are given, wherever possible.
- Programming exercises to enable students to test programming skills.
- Summary provided for quick revision of concepts.
- Review questions and exercises to enhance application skills.

Must buy for students of M.Tech., M.C.A., M.Sc., (Mathematics and Computer Science) Engineering, BCA, BIT, B.Sc., (Mathematics and Computer Science), C-DAC, DOEACC-'O' Level, 'A' Level and other diploma courses.

ABOUT THE AUTHOR

J. B. Dixit is a high profile author. He has an avid interest in seeing students become well educated - especially in Information Technology, Programming and Mathematics. He spends his time watching what's happening in business and society and on college/university campuses and listening to the view expressed by lecturers, students and other participants in the Computer and mathematics revolution. He then tries to translate those observations into meaningful language that can be best understood by students.

Over the past decade, he has authored more than 25 quality books, most of them on Computers, Information Technology and Mathematics for Engineering, level courses.

He has been teaching Engineering M.Tech., M.C.A., M.B.A., M.Sc., B.C.A., B.I.T., B.B.A. and other diploma course students for the last fifteen years. Many of his students are very well placed in IT industry all over the world (including Microsoft Corporation, USA). A logically sound and experienced teacher, recipient of various honours and scholarships, he has few more books in the pipeline for various Engineering and Degree level courses.

FIREWAL MEDIA

ISBN 978-81-318-0695-1