2.1 INTRODUCTION

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. 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 compiler implementations usually use hash tables to look up identifiers.

Data structure is a representation of the logical relationship between data elements. In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

2.2 WHAT IS AN ALGORITHM?

Once a data structure for a particular application is chosen, an algorithm must be developed that manipulates the related data items stored in it.

Every problem solution starts with a plan. That plan is called an algorithm.

An algorithm is a plan for solving a problem.

Our algorithm development process consists of five major steps:

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

Such an algorithm should have the following features:

1. It should be free of ambiguity.
2. It should be concise.
3. It should be efficient.

An *algorithm* is a precise plan for performing a sequence of actions to achieve the intended purpose. Each action is drawn from a well-understood selection of actions on data.

Definition: An *algorithm* is a step-by-step finite sequence of instructions, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem. In addition every algorithm must satisfy the following criteria:

(i) **Input:** there are zero or more quantities which are externally supplied;

(ii) **Output:** at least one quantity is produced;

(iii) **Definiteness:** each instruction must be clear and unambiguous;

(iv) **Finiteness:** if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

(v) **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 as in (iii), but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy condition (iv). 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.

Therefore, *an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input.*

2.3 TYPES OF ALGORITHM

Basically the problems to be solved with the help of computer are of three types. So, Algorithm is also divided into following three types:

1. Sequential
2. Selective
3. Iterative

Sequential

In this approach the steps of algorithm will start from step1 and execution will start from starting step to the last step in which it terminates in a sequential manner that is top to down.

Example: Write an algorithm to input two numbers and print its sum as output.

Solution:

Step 1: Input two numbers and store it into A and B
Step 2: Add A and B and store result into sum
Step 3: Print sum
Step 4: Stop

Selective

In this approach, selective problems used, where any condition is arises in any step, so the algorithm must be capable to satisfy both part of condition, whether it is true or false.

Example: Input two numbers find out the largest number

Solution:

Step 1: Read two numbers let A and B
Step 2: If A is greater than B, then store A into L otherwise store B into L
Step 3: Print "Larger = ", L
Step 4: Stop

Iterative

In this approach, one or more steps are repeating number of times, so such types of iterative problems are using repeating steps.

Example: Generate N Natural Numbers and find out their sum.

Solution:

Step 1: Read the range let N

Step 2: Let C= 0, S=0

Step 3: Add 1 to C
Add C to S

Step 4: Print C

Step 5: If C < N then goto Step 3.

Step 6: Print "Sum=", S

Step 7: Stop

2.4. DESIRABLE ATTRIBUTES FOR ALGORITHMS

Although an algorithm may satisfy the criteria of the previous section by being precise, unambiguous, deterministic, and finite, it still may not be suitable for use as a computer solution to a problem. The basic desirable attributes that an algorithm should meet are explained below:

- **Generality:** An algorithm should solve a class of problems, not just one problem. For example, an algorithm to calculate the average of four values is not as generally useful as one that calculates the average of an arbitrary number of values.
- **Good Structure:** This attribute applies to the construction of the algorithm. A well-structured algorithm should be created using good building blocks that make it easy to
  - Explain,
  - Understand,
  - Test, and
  - Modify it.

  The blocks, from which the algorithm is constructed, should be interconnected in such a way that one of them can be easily replaced with a better version to improve the whole algorithm, without having to rebuild it.
- **Efficiency:** An algorithm's speed of operation is often an important property, as is its size or compactness. Initially, these attributes are not important concerns. First, we must create well-structured algorithms that carry out the desired task under all conditions. Then, and only then, do we improve them with efficiency as an objective.
- **Ease of Use:** This property describes the convenience and ease with which users can apply the algorithm to their data. Sometimes what makes an algorithm easy to understand for the user also makes it difficult to design for the designer (and vice versa).
- **Elegance:** This property is difficult to define, but it appears to be connected with the qualities of harmony, balance, economy of structure and contrast, whose contribution to beauty in the arts is prized. Also, as with the arts, we seem to be able to "know beauty when we see it" in algorithms. Sometimes the term 'beauty' is used for this attribute of an algorithm.

  Other desirable properties, such as ***robustness*** (resistance to failure when presented with invalid data) and ***economy*** (cost-effectiveness), will only be touched upon in this

chapter. Not because these properties are unimportant, but because the basic attributes of an algorithm should be understood first.

2.5. ALGORITHM DESIGN TECHNIQUES

There are two approaches for algorithm design; they are top-down and bottom-up algorithm design

(a) Top-down algorithm design

The principle of top-down design states that a program should be divided into a main module and its related modules. Each module should also be divided into sub-modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary processes that are basically understood and cannot be further subdivided.

Top-down algorithm design is a technique for organizing and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

In C, the idea of top-down design is done using functions. A C program is made of one or more functions, one and only one of which must be named main. The execution of the program always starts and ends with main, but it can call other functions to do special tasks.

(b) Bottom-up algorithm design

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and builds them into more complex structures, ending at the top.

The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the main function is tested.

For a given problem, there are many ways to design algorithms for it e.g. insertion sort is an incremental approach, Merge sort is a divide and conquer approach. The following is a list of several popular design approaches:

1. Incremental approach
2. Divide and Conquer approach
3. Greedy approach
4. Dynamic programming approach
5. Backtracking approach
6. Branch and bound approach
7. Randomized approach

Example: Suppose we want to find the average of three numbers, the algorithm is as follows:

Step 1: Read the numbers a, b, c
Step 2: Compute the sum of a, b and c
Step 3: Divide the sum by 3
Step 4: Store the result in variable d
Step 5: Print the value of d
Step 6: End of the program

Example: Write an algorithm to find the area of the triangle. Let b, c be the sides of the triangle ABC and A the included angle between the given sides.

Step 1: Input the given elements of the triangle, namely sides b, c and angle between the sides A.
Step 2: Area = (1/2) *b*C* sin A
Step 3: Output the Area
Step 4: Stop

Example: Write an algorithm which will test whether a given integer value is prime or not.

Step 1: k ← 2
Step 2: read N
Step 3: MAX SQRT (N)
Step 4: While k< = MAX
do if (k* (N/k) = N
then
go to step 7
k k + 1
Step 5: Write "number is prime"
Step 6: go to step 8
Step 7: Write "number is not a prime"
Step 8: end

Example: Write algorithm to find the factorial of a given number N

Step 1: FAC 1
Step 2: I 0
Step 3: read N
Step 4: While I< N
do I I + 1
FAC FAC* I
Step 5: Write "Factorial of", N, "is", FAC
Step 6: end

2.6. FLOWCHART

A flowchart is a step by step diagrammatic representation of the logic paths to solve a given problem or a flowchart is a pictorial representation of the algorithm. It represents the steps involved in the procedure and shows the logical sequence of processing using boxes of different

shapes. The instruction to be executed is mentioned in the boxes.These boxes are connected together by solid lines with arrows, which indicate the flow of operation.

A flowchart when translated in to a proper computer language, results in a complete program.

2.6.1 Advantages of Flowcharts

1. The flowchart shows the logic of a problem displayed in pictorial way which felicitates easier checking of an algorithm.
2. The flowchart is good means of communication to other users. It is also a compact means of recording an algorithm solution to a problem.
3. The flowchart allows the problem solver to break the problem into parts. These parts can be connected to make master chart.
4. The flowchart is a permanent record of the solution which can be consulted at a later time.

2.6.2 Symbols used in Flowcharts

The symbols that we make use while drawing flowcharts as given below are as per conventions followed by International Standard Organization (ISO).

Oval: Rectangle with rounded sides is used to indicate either START/ STOP of the program.

Fig. 2.1

Input and Output indicators: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are represented in these Parallelograms.

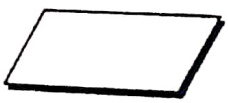

Fig. 2.2

Process Indicators: Rectangle is used to indicate any set of processing operation such as for storing arithmetic operations.

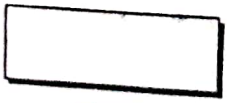

Fig. 2.3

Decision Makers: The diamond is used for indicating the step of decision making and therefore, known as decision box. Decision boxes are used to test the conditions or ask questions and depending upon the answers, the appropriate actions are taken by the computer. The decision box symbol is

Fig. 2.4

shapes. The instruction to be executed is mentioned in the boxes.These boxes are connected together by solid lines with arrows, which indicate the flow of operation.

A flowchart when translated in to a proper computer language, results in a complete program.

2.6.1 Advantages of Flowcharts

1. The flowchart shows the logic of a problem displayed in pictorial way which felicitates easier checking of an algorithm.
2. The flowchart is good means of communication to other users. It is also a compact means of recording an algorithm solution to a problem.
3. The flowchart allows the problem solver to break the problem into parts. These parts can be connected to make master chart.
4. The flowchart is a permanent record of the solution which can be consulted at a later time.

2.6.2 Symbols used in Flowcharts

The symbols that we make use while drawing flowcharts as given below are as per conventions followed by International Standard Organization (ISO).

Oval: Rectangle with rounded sides is used to indicate either START/ STOP of the program.

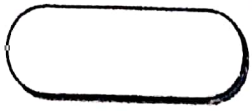

Fig. 2.1

Input and Output indicators: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are represented in these Parallelograms.

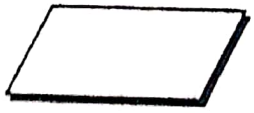

Fig. 2.2

Process Indicators: Rectangle is used to indicate any set of processing operation such as for storing arithmetic operations.

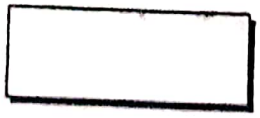

Fig. 2.3

Decision Makers: The diamond is used for indicating the step of decision making and therefore, known as decision box. Decision boxes are used to test the conditions or ask questions and depending upon the answers, the appropriate actions are taken by the computer. The decision box symbol is

Fig. 2.4

Flow Lines: Flow lines indicate the direction being followed in the flowchart. In a flowchart, every line must have an arrow on it to indicate the direction. The arrows may be in any direction.

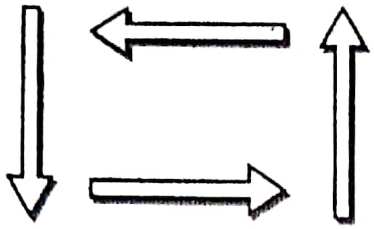

Fig. 2.5

On-Page Connectors: Circles are used to join the different parts of a flowchart and these circles are called on-page connectors. The uses of these connectors give a neat shape to the flowcharts. In complicated problems, a flowchart may run in to several pages. The parts of the flowchart on different pages are to be joined with each other. The parts to be joined are indicated by the circle.

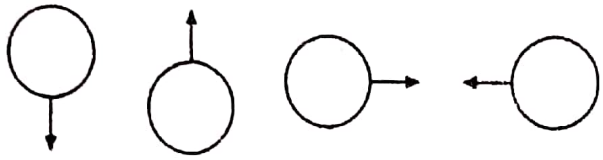

Fig. 2.6

Off-page Connectors: This connector represents a break in the path of flowchart which is too large to fit on a single page. It is similar to on-page connector. The connector symbol marks where the algorithm ends on the first page and where it continues on the second.

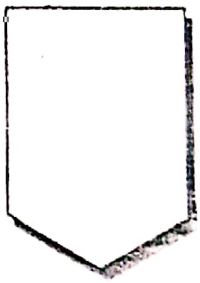

Fig. 2.7

Example:Draw a flowchart to add two numbers entered by user.

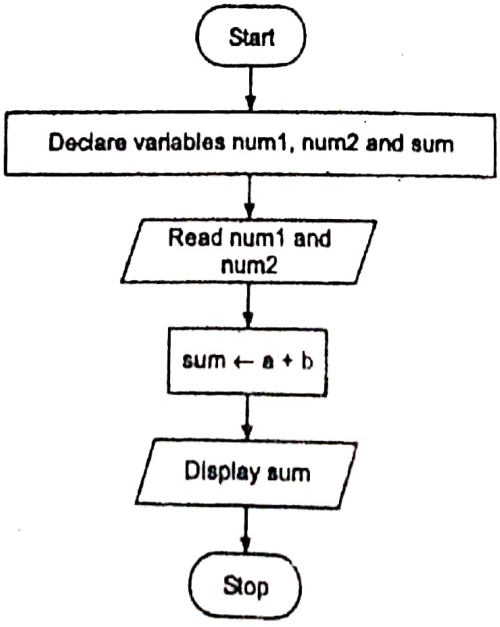

Fig. 2.8

Example: Convert Temperature from Fahrenheit (?) to Celsius (?)

Algorithm:

Step 1: Read temperature in Fahrenheit,

Step 2: Calculate temperature with formula C=5/9*(F-32),

Step 3: Print C,

Flowchart:

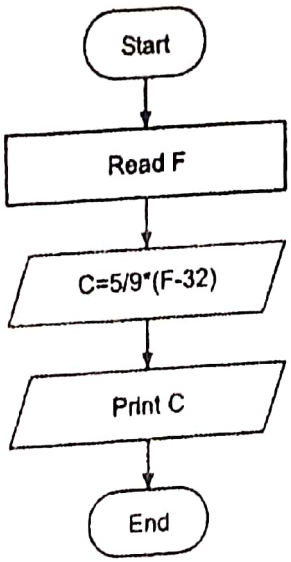

Fig. 2.9.

Example: Make a flow chart to input previous, current reading and rate/unit of electric consumption. Calculate and print the unit consumed and payable amount.

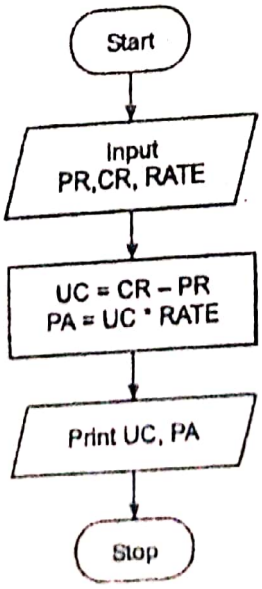

Fig. 2.10.

***Example:** Find out Largest and Smallest of any three Numbers.*

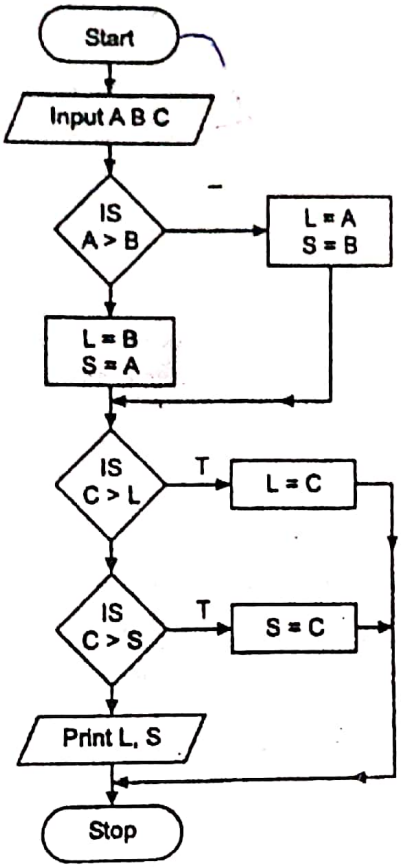

Fig. 2.11.

***Example:** Input any Number; find out factorial of that Number.*

The factorial of any number is this product of all Natural number from 1 to that number.

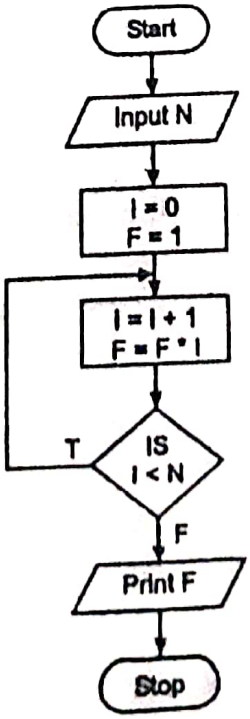

Fig. 2.12.

2.6.3 Limitations of Flowcharts

- The major disadvantage in using flowcharts is that when a program is very large, the flowcharts may continue per many pages, making them hard to follow.
- Drawing flowcharts are possible only if the problem solving logic is straight forward and not very lengthy.
- Due to its symbolic nature, any changes or modification to a flowchart usually requires redrawing the entire logic again, and redrawing a complex flowchart is not a simple basic.

2.7 PSEUDO-CODES

We usually present algorithms in the form of some pseudo-code, which is normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language. Actually, pseudo code is an artificial and informal language that helps programmers develops algorithms. It is a "text-based" detail (algorithmic) design tool. There is no standard convention for writing pseudo-code.

> **Note:** Pseudocode is made up of two words ***pseudo*** and ***code***. Pseudo means limitation i.e. not as rigorous as a programming language and code refers to instructions, written in a programming language.

Examples:

If student's grade is greater than or equal to 60

```
        PRINT "passed"
else
        PRINT "failed"
```

Pseudo-code uses some keywords to denote programming processes. Some of them are:

- Input: READ, OBTAIN, GET and PROMPT
- Output: PRINT, DISPLAY AND SHOW
- Compute: XOMPUTE, CALCULATE AND DETERMINE
- Initialize: SET AND INITIALISE
- Add One: INCREMENT

Guidelines of Pseudo-code:

- Statements should be written in simple English and should be programming language independent.
- Steps must be understandable and when the steps are followed it must produce a solution to the specified problem
- Pseudo-codes should be concise.
- Each instruction should be written in a separate line and each statement in pseudo-code should express just one action.
- Capitalize keywords such as READ, PRINT and so on.
- Each set of instructions is written from top to bottom, with only one entry and one exit.
- Flowcharts for easy transition from design to coding in programming language.

Benefits of Pseudo-code:

1. Since it is language independent, it can be used by most programmers.
2. It allows the developer to express the design in plain natural language.
3. It is easier to develop a program from a pseudo-code than with a flowchart.
4. It is easy to translate pseudo-code into a programming language by less experienced programmers.
5. The use of words and phrases in pseudo-code, which are in line with basic computer operations, simplifier the translation from the pseudo-code algorithm to a specific programming language.
6. It's simple structure and readability makes it easier to modify.

Limitations of Pseudo-code

- The main disadvantage of using pseudo-code is that it does not provide visual representation of the program's logic
- Pseudo-code cannot be complied nor executed and more are not real formatting or syntax rules.

Example: ***Find area of a triangle***

Pseudo-code:

```
Set initial a,b,c
Read the value of a,b,c
To calculate three sides of triangle using formula
    s = (a+b+c)/2
To find area of a triangle using formula
    Area =sqrt (s*(s - a)*(s-b)*(s-c))
WRITE the output Area
Stop
```

Example: ***Write a program to find the largest of three numbers.***

Pseudo-code:

```
Set initial a, b, c
READ the value for a, b, c
IF (a>b) and (a>c) THEN
    WRITE 'a is largest'
Else, IF (b>c) THEN
    WRITE 'b is largest'
ELSE
    WRITE 'c is largest'
END IF
Stop.
```

Example: ***Write a program to find the given year is a leap year or not.***

Pseudo-code:

```
Set initial value of year.
READ the value of year.
```

```
IF (year % 4 = 0) Then
    WRITE year is a leap year
ELSE
    WRITE the year is not a leap year.
ENDIF
STOP
```

2.8. CODING (FROM ALGORITHMS TO PROGRAMS)

An algorithm is a autonomous step-by-step set of operations to be performed to solve a specific problem or a class of problems. A computer **program** is a sequence of instructions that fulfill the rules of a specific **programming** language, written to perform a specified task with a computer.

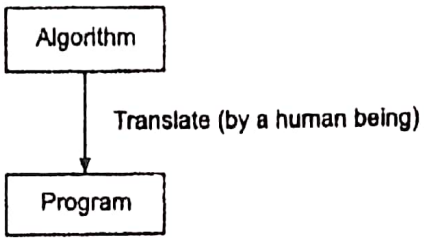

Fig. 2.13.

Once the design process is complete, the actual computer program is written, i.e. the instructions are written in a computer language. Coding is generally a very small part of the entire program development process and also a less time consuming activity in reality. In this process all the syntax errors i.e. errors related to spelling, missing commas, undefined labels, etc. are eliminated. For effective coding some of the guidelines which applied are:

- Use of meaningful names and labels of variables,
- Simple and clear expressions,
- Modularity with emphasis on making modules generalized,
- Making use of comments and indenting the code properly,
- Avoiding jumps in the program to transfer control.

2.8.1 Documenting the Program

During both the algorithm development and program writing stages, explanations called documentation are added to the code. It helps users as well as programmers understand the exact processes to be performed.

2.8.2 Testing and Debugging the Program

Program errors are known as *bug*. Process of detecting and correcting these errors is called *debugging*. Testing is the process of making sure that the program performs the intended task.

Types of Program Errors

Syntax errors: Syntax errors occurs when the rules or syntax of the programming language are not followed e.g. incorrect punctuation, undefined terms, incorrect word sequence and misuse of terms. These errors are detected by a language processor.

Logic errors: Logical error occurs due to errors in planning a program's logic. Such errors cause the program to produce incorrect output. These errors cannot be detected by a language processor.

- The program must be free of syntax errors.
- The program must be free of logic errors.
- The program must be reliable (produces correct results).
- The program must be robust (i.e. able to detect execution errors).

EXERCISES

1. What is an Algorithm?
2. Write all the characteristics of an Algorithm.
3. Write an Algorithm to find out sum and product of any two Numbers.
4. Write an Algorithm to check that whether a given number is perfect or not.
5. Write an Algorithm to count number of pass and fail student is a class of N students by inputting test marks.
6. What is flowchart, how it is different than Algorithm?
7. Make a flowchart to check whether a given number is prime or not prime.
8. Make a flowchart and Pseudo-code to Generate Fibonacci series terms upto N.
9. Make a flowchart and Pseudo-code to count Number of voters in a city of population N.
10. Make a flowchart to find out sum of following series:

    $S = 1 + 11 + 111 + 1111 + 11111$

4 Fundamentals of C Language

4.1 INTRODUCTION

C is a popular general purpose programming language. C language has been designed and developed by *Dennis Ritchie* at bell laboratories in *1972*. The C language is derived from the *B language*, which was written by *Ken Thompson* at AT&T Bell laboratories. The B language was adopted from a language called BCPL (Basic Combined Programming Language), which was developed by *Martin Richards* at Cambridge University. In 1982, a committee was formed by ANSI (American National Standards Institute) to standardize the C language. Finally in 1988, the standard for C language was introduced known as ANSI C.

C language is *middle - level computer language*. It reduces the gap between high level language and low - level language so, it is known as ***middle level language***.

C is a *structured language*. It is similar in many ways to other structural languages such as Pascal and FORTRAN. A structured language allows variety of programs in small modules. The source code for the Linux operating system is coded in C. C runs under a number of operating systems including MS - DOS.

The programs written in C are portable i.e. programs written for one type of computer or operating system can be run\on another type of computer or operating system.

C has become a popular programming language because of its many features . Some important features are as follows:

- Reliable, simple, and easy to use
- General purpose language
- Has qualities of high-level programming language with efficiency of assembly language
- Helps in development of system software
- Supports user-defined data types
- Supports modular and structured programming concepts
- Supports a rich library of functions
- Supports pointers with pointer operations
- Supports low-level memory and device access
- Small and concise language
- Standardized by several international standards body

4.1 C PROGRAM

A C program can vary from three lines to millions of lines and it should be written into one or more text files with extension ".c"; for example, **hello.c**. We can use "vi", "vim" or any other text editor to write your C program into a file.

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World".

```
#include <stdio.h>
int main()
{
/* My first program in C */
printf("Hello, World! \n");
return 0;
}
```

1. Lines beginning with **#** are called **"preprocessor directives"** which are processed by the compiler before starting compiler. **#include<stdio.h>** tells the preprocessor to include the **standard Input/output Library/header (<stdio.h>)** in this program..
2. The next line **intmain()** is the main function where program execution begins.
3. The next line **/*...*/** will be ignored by the compiler. Such lines are called comments in the program.
4. The next line **printf(...)** is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
5. The next line **return 0;** terminates **main()**function and returns the value 0

Note that the "\n" is not printed to the screen. The expressions inside printf starting with "\" (Back Slashes) are called **Escape Sequences**. Escape Sequences are not printed screen but only used to control way of the printing. Here, "\n" is the newline escape sequence indicating that cursor will go to the next line at this point. There are other escape sequences available to do the printing in more flexible way.

Escape sequence	Description

4.3 STRUCTURE OF A C PROGRAM

The basic components of a C program are:

- main() function
- pair of curly braces { }
- declarations and statements
- user-defined functions

The complete structure of C program is as follows:

```
Include header file section (preprocessor statements)
Global declaration section
/ * comments */
main() /* Function name */
{
/ * comments * /
Declaration part
Executable part
}
User-defined functions
(
Statements
;
```

Include header file section

C program depends upon some header files for function definition that are used in program. Each header file by default is extended with **.h**. The header file should be included using **# include** directive.

Preprocessor statements begin with # symbol and are also called preprocessor directives. These statements direct the C preprocessor to include header files and also symbolic constants into a C program. Some of the preprocessors statements are given below:

# Include <stdio.h>	for the standard input/output function
# Include " stdio.h>	for file inclusion of header file test
#define NULL 0	for defining symbolic constant NULL=0

Braces

Every C program uses a pair of curly braces ({,}). The left brace indicates the beginning of main() function. On the other hand, the right brace indicates the end of main() function. The braces can also be used to indicate the beginning and end of user defined functions and compound statements.

Declarations

It is a part of the C program where all the variables, arrays, functions, etc. used in the C program are declared and may be initialized with their basic data types.

Statements

These are instructions to the computer to perform specific operations. They may be input-output statements, arithmetic statements, control statements and other statements. They also include comments. Comments are explanatory notes on some instructions. The statements to be commented on must be enclosed within /* and */ Comment statements are not compiled and executed.

User-Defined Functions

These are subprogram. Generally, a sub-program is a function. And they contain a set of statements to perform a specific task. These are written by the users, hence the name user defined functions. They may be written before or after main() function.

Comments

Comments are not necessary in the program. Comments are to be inserted by the programmer. Comments are nothing but same kind of statement which are placed between the delimiters logo. The compiler does not execute comments.

C permits different forms of main statement. Following forms are used:

- main()
- main(void)
- void main()
- int main()
- void main(void)
- int main(void)

Empty pair of parentheses shows that the function has no arguments. This may be clearly indicated by using the keyword void inside the parentheses. We can also specify the keyword int or void before the word main. void means that the function does not return any information to the operating system and int means that the function returns an integer value to the operating system. So, when int is used program must be "return 0".

4.4 PROGRAMMING RULES

1. All statements should be written in lower case letter. Upper case letters are only used for symbolic constants.
2. Bank spaces may be inserted between the words.
3. The user can also write

White spaces are Blank space, horizontal tab, carriage return, new line and form feed. The compiler ignores white spaces except they are a part of a string constant. White spaces may be used to separate words, but are forbidden between the characters of keywords and identifiers.

4.4 C TOKENS

The basic atomic units of a C program are called C tokens. There are six types of tokens in C as follows

- keywords,
- Identifiers
- Constants
- Strings
- Operators
- Special symbols

C programs are written using these tokens and the syntax of the C language

4.5 KEYWORDS

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. A constant is an entity that doesn't change, whereas a variable is an entity that may change.

Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. The keywords are also called 'Reserved words'

There are only 32 keywords available in C.

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

Additional keywords for Borland c are as follows

Asm	Cdecl	Far	Huge	Interrupt	Near	pascal

4.6 IDENTIFIERS

functions and arrays. They are user-defined names consisting

Rules for forming identifier name:

- The first character must be an alphabet (upper case or lower case) or an underscore.
- All succeeding characters must be either letters or digits.
- Upper case and lower case identifiers are different in C.
- No special character or punctuation symbols are allowed except the underscore "_".
- No two successive underscores are allowed.
- Keywords should not be used as identifier.

4.10 CONSTANTS

The data item which remain always fixed throughout the execution of the program, which can't be modified or vary is known as constant. There are following different types of constants available in 'C' language:

C constants can be divided into two major categories:

1. Numerical Constants
2. Character Constants

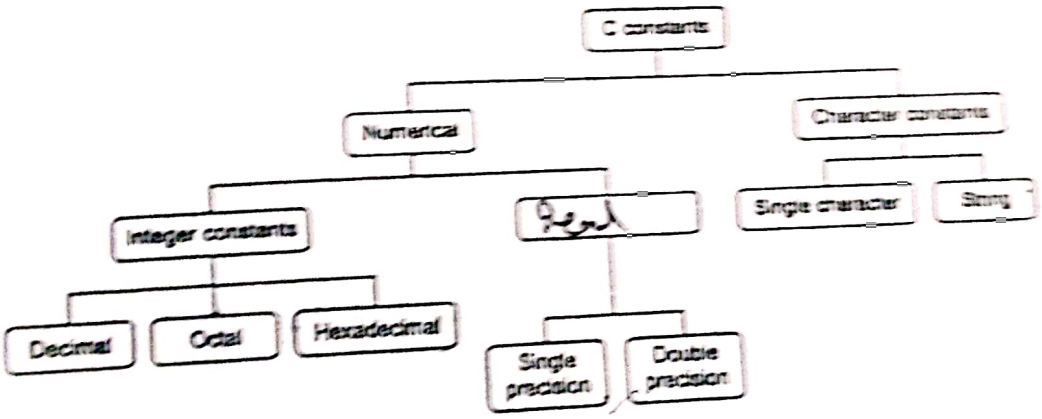

Fig. 4.1.

Numerical Constants

(a) Integer constants

These are the sequence of numbers from 0 to 9 without decimal points or fractional part or any other symbols. It requires minimum two bytes and maximum four bytes integer constants could either be positive or negative or may be zero.

For example: 10, 20, +70, -5, etc.

Rules for Constructing Integer Constants:

- An integer constant must have at least one digit.
- It can be either positive or negative.
- If no sign precedes an integer constant it is assumed to be positive.
- No commas or blanks are allowed within an integer constant.
- It must not have a decimal point.
- The allowable range for integer constants is -32768 to 32767.

Character Constant

(a) Single character constants: A character constant is a single character. Characters are also represented with a single digit or a single special symbol or white space enclosed within a pair of single quote marks.

For example: 'a', '8', etc.

(b) String constants: String constants are sequence of characters enclosed within double quote marks. The string may be a combination of all kinds of symbols.

For example: "Parv", "India", "786", "a".

Rules for Constructing Character Constants:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, 'A' is a valid character constant, whereas 'A' is not.
- The maximum length of a character constant can be 1 character.

4.11. C VARIABLES

In any program we typically do lots of calculations. The calculated values are stored in computer memory. Computer memory consists of millions of cells. The calculated values are stored in these memory cells. To make the retrieval and usage of these values straightforward these memory cells (also called memory locations) are given names. Since the value stored in each location may change the names given to these locations are called variable names. So, variable names are names given to locations in memory. These locations can contain integer, real or character constants. A particular type of variable can hold only the same type of constant. For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

Rules for Constructing Variable Names

- A variable name is any combination of 1 to 31 alphabets, digits or underscores.
- The first character in the variable name must be an alphabet or underscore.
- All succeeding characters consists of letters and digits
- No commas or blanks are allowed within a variable name.
- Keywords should not be used as variables.
- No special symbol other than an underscore can be used in a variable name.
- **The maximum allowable length of a variable name is 31 characters.**
- Always choose an appropriate variable name that makes proper sense to the user

The table shows the valid and invalid variable names.

Valid Variables	Invalid Variables
1. marks	1. 8ab
2. TOTAL_MARK	2. TOTAL MARK
3. gross_salary_2001	3. gross-salary-2001
4. area_of_circle()	4. Area_ _ of _ _ salary
5. num[20]	

4.12 DATA TYPES

Data types indicate the type of data that a variable can hold. The data may be numeric or not numeric in nature. Each data type requires different amounts of memory and has some specific operations which can be performed over it. In C, the data types are categorized into:

1. Built in data type
2. Derived data type
3. User-defined data type

Built-in data type (Fundamental Data Types)

All C compilers support 5 built-in data types such as:

1. Integer **int**
2. Character **char**
3. Floating Point **float**
4. Double Precision Floating Point **double**
5. Void Data Type **void**(used for function when no value is to be return)

Derived Data Type

Those data types which are derived from fundamental data types are called derived datatypes. There are basically three derived data types:

1. Array: A finite collection of data of same types or homogenous data type.
2. String: An array of character type.
3. Structure: A collection of related variables of the same or different data types.

User defined Data type

The user defined data types enable a program to invent his own data types and define what values it can take on. Thus, these data types can help a programmer to reducing programming

Data Type	Minimum Storage Allocated	Used for Variables that can contain
int	2 bytes (16 bits)	integer constants in the range -32768 to 32767
short	2 bytes (16 bits)	integer constants in the range -32768 to 32767
long	4 bytes (32 bits)	integer constants in the range -2147483648 to 2147483647
float	4 bytes (32 bits)	real constants with minimum 6 decimal digits precision
double	8 bytes (64 bits)	real constants with minimum digits precision 10 decimal
char	1 byte (8 bits)	character constants
enum	2 bytes (16 bits)	Values in the range -32768 to 32767
void	No storage allocated	No value assigned

As, we know the primary data types could be of three varieties-char, int, and float. The primary data types themselves could be of several types. For example, a char could be an unsigned char or a signed char or an int could be a short int or a long int.

INTEGER DATA TYPES

C offers a variation of the integer data type such as short and long integer values.

Short Integer	Long Integer
Occupies 2 bytes in memory	Occupies 4 bytes in memory
Range: -32,768 to +32,767	Range : -2147483648 to +2147483647
Program runs faster	long integers cause the program to run a bit slower
Format specifies is %d or %i	Format specifies is %ld
Example : Short int a=7;	Example: long int a;

Sometimes, we know in advance that the value stored in a given integer variable will always be positive. In such a case we can declare the variable to be unsigned. Declaring an integer as unsigned almost doubles the size of the largest possible value that it can otherwise take. Like an unsigned int, there also exists a short unsigned int and a long unsigned int. By default a short int is a signed short int and a long int is a signed long int.

Signed Integer	Unsigned Integer
Occupies 2 bytes in memory	Occupies 2 bytes in memory
Range: -32,768 to 32,767	Range : -0 to 65535
Format specifies is %d or %i	Format specifies is %u
Example :	Example:
int a=0;	unsigned long b;
long int b=5;	Unsigned short int c;

3.11 SYNTAX ERRORS

Sometimes, when you tell your Integrated Development Environment (IDE) to translate a program from source code to object code, you get some error messages. The translator (whether it is a compiler or an interpreter) cannot convert the code. This is because the rules of the programming language you are using have been broken.

A syntax error is an error in the source code of a program. Since computer programs must follow strict syntax to compile correctly, any aspects of the code that do not conform to the syntax of the programming language will produce a syntax error. Syntax errors are small grammatical mistakes, sometimes limited to a single character. It might be that you have not spelled a **'reserved word'** correctly. It might be that you haven't used a **'reserved word'** in the correct way. (A **'reserved word'** is a word that is used by the programming language - you are not allowed to use it as an identifier.). A missing semicolon at the end of a line or an extra bracket at the end of afunction may produce a syntax error.

Five common syntax errors are:

- Spelling mistakes.
- Missing out quotes.
- Missing out brackets.
- Missing out a colon or semicolon at the end of a statement.
- Using upper case characters in keywords, e.g., IF instead of if.

3.12 LOGIC ERRORS

Syntax errors occur when a program does not conform to the grammar of a programming language, and the compiler cannot compile the source file. Logic errors occur when a program does not do what the programmer expects it to do.

A logic error (or logical error) is a 'bug' or mistake in a program's source code that results in incorrect or unexpected behaviour. It is a type of runtime error that may simply produce the wrong output or may cause a program to crash while running.

For example, if a programmer wrote this:

MonthsInYear = 13

Average = Total / MonthsInYear

and compiled it, it would compile and run okay. It would give you an answer. However, it would give an incorrect answer (there are only 12 months in a year).

So, a logical error occurs due to the poor understanding of the problem

3.13 SOURCE CODE

In computing, source code is any collection of code, probably with comments, written using a human-readable programming language, usually as plain text. The source code of a program is specially designed to help the work of computer programmers, who specify the actions to be performed by a computer mostly by writing source code.Source code is primarily used as input to the process that produces an executable program.

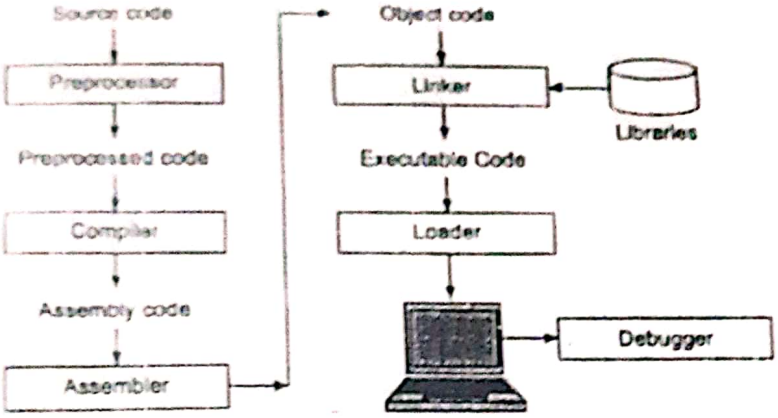

Fig. 3.4.

The source code is often transformed by an assembler or compiler into binary machine code understood by the computer.

3.14 OBJECT CODE

In computing, object code or object module is the product of a compiler. In a general sense object code is a sequence of statements or instructions in a computer language, usually a machine code language (i.e., binary) or an intermediate language such as register transfer language (RTL).

Most programs are written with source code logically divided into multiple source files. Each source file is compiled independently into a corresponding "object" file of partially-formed machine code known as **object code**. At a later time these "object files" are "linked" together to form an **executable file**.

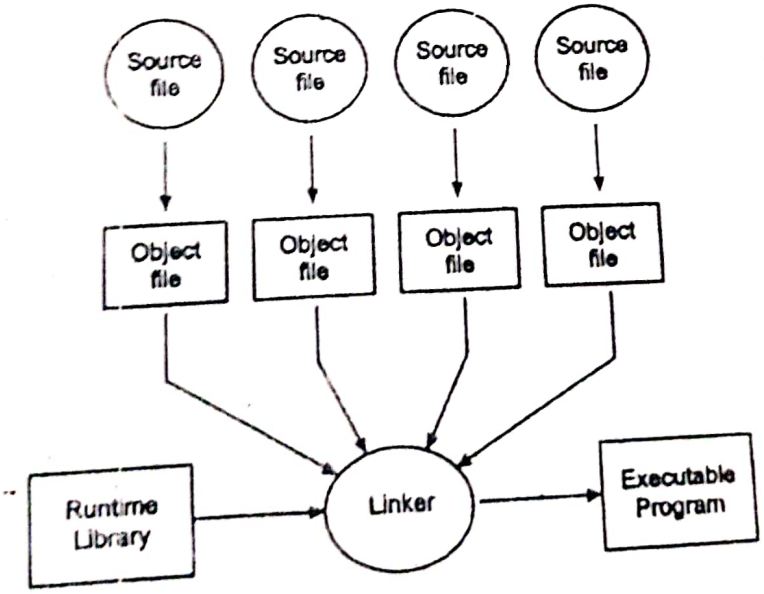

Fig. 3.5.

3.15 EXECUTABLE CODE

The linker, as a final phase of compilation, will read all of the object files, resolve references between them, perform the final code layout in memory that determines the addresses for all the blocks of code and data, fix up all the placeholder addresses with real addresses, and write out the executable file.

Object files have a lot in common with executable files (table of contents, blocks of machine instructions and data, and debugging information). However, the code isn't ready to run. It is full of incomplete references to subroutines outside itself, and as such, many of the machine instructions have only placeholder addresses.

An executable file is a complete program that can be run directly by an operating system (in conjunction with shared libraries and system calls). The file generally contains a table of contents, a number of code blocks and data blocks, auxiliary data such as the memory addresses at which different blocks should be loaded, which shared libraries are needed, the entry point address, and sometimes a symbol table for debugging. An operating system can run an executable file more or less by loading blocks of code and data into memory at the indicated addresses and jumping to it.

In simple words, executable code is the code in machine language (binary code)which is directly understood by the computer and executes it without further simplification needed while Object code is the code produced by a compiler or interpreter.

EXERCISES

1. What is the need for programming languages?
2. What are the different types of programming languages? Explain them.
3. List the merits and demerits of:
   (*a*) Assembly Language
   (*b*) High level Language
4. What is procedure oriented and object oriented languages?
5. What is Structure Programming Language?
6. Define the main concept of data type.
7. What is variable?
8. Write short notes on the following:
   (*a*) ASCII code
   (*b*) EBCEIC code
9. Why binary system is important in representing data in computer? Explain.
10. Discuss the various steps in the process of compilation.
11. Define:
    (*a*) source code
    (*b*) object code
    (*c*) executable code
12. Distinguish between syntax errors and logical errors.