UNIT 1

Introduction to Problem Solving Techniques

Structure

Learning Objectives

- To understand the concept of Problem solving
- To understand steps involved in algorithm development
- To understand the concept of Algorithm
- Develop Algorithm for simple problem
- To understand the concept of Flowchart development
- Draw the symbols used in Flowcharts

1.0 Introduction

A computer is a very powerful and versatile machine capable of performing a multitude of different tasks, yet it has no intelligence or thinking power. The intelligence Quotient (I.Q) of a computer is zero. A computer performs many tasks exactly in the same manner as it is told to do. This places responsibility on the user to instruct the computer in a correct and precise manner, so that the machine is able to perform the required job in a proper way. A wrong or ambiguous instruction may sometimes prove disastrous.

In order to instruct a computer correctly, the user must have clear understanding of the problem to be solved. A part from this he should be able to develop a method, in the form of series of sequential steps, to solve it. Once the problem is well-defined and a method of solving it is developed, then instructing he computer to solve the problem becomes relatively easier task.

Thus, before attempt to write a computer program to solve a given problem. It is necessary to formulate or define the problem in a precise manner. Once the problem is defined, the steps required to solve it, must be stated clearly in the required order.

1.1 Procedure (Steps Involved in Problem Solving)

A computer cannot solve a problem on its own. One has to provide step by step solutions of the problem to the computer**.** In fact, the task of problem solving is not that of the computer. It is the programmer who has to write down the solution to the problem in terms of simple operations which the computer can understand and execute.

In order to solve a problem by the computer, one has to pass though certain stages or steps. They are

1. Understanding the problem
2. Analyzing the problem
3. Developing the solution
4. Coding and implementation.

1. Understanding the problem: Here we try to understand the problem to be solved in totally. Before with the next stage or step, we should be absolutely sure about the objectives of the given problem.

2. Analyzing the problem: After understanding thoroughly the problem to be solved, we look different ways of solving the problem and evaluate each

of these methods. The idea here is to search an appropriate solution to the problem under consideration. The end result of this stage is a broad overview of the sequence of operations that are to be carries out to solve the given problem.

3. Developing the solution: Here the overview of the sequence of operations that was the result of analysis stage is expanded to form a detailed step by step solution to the problem under consideration.

4. Coding and implementation: The last stage of the problem solving is the conversion of the detailed sequence of operations in to a language that the computer can understand. Here each step is converted to its equivalent instruction or instructions in the computer language that has been chosen for the implantation.

1.2 Algorithm

Definition

A set of sequential steps usually written in Ordinary Language to solve a given problem is called **Algorithm**.

It may be possible to solve to problem in more than one ways, resulting in more than one algorithm. The choice of various algorithms depends on the factors like reliability, accuracy and easy to modify. The most important factor in the choice of algorithm is the time requirement to execute it, after writing code in High-level language with the help of a computer. The algorithm which will need the least time when executed is considered the best.

Steps involved in algorithm development

An algorithm can be defined as **"a complete, unambiguous, finite number of logical steps for solving a specific problem "**

Step1. Identification of input: For an algorithm, there are quantities to be supplied called input and these are fed externally. The input is to be indentified first for any specified problem.

Step2: Identification of output: From an algorithm, at least one quantity is produced, called for any specified problem.

Step3 : Identification the processing operations : All the calculations to be performed in order to lead to output from the input are to be identified in an orderly manner.

Step4 : Processing Definiteness : The instructions composing the algorithm must be clear and there should not be any ambiguity in them.

Step5 : Processing Finiteness : If we go through the algorithm, then for all cases, the algorithm should terminate after a finite number of steps.

Step6 : Possessing Effectiveness : The instructions in the algorithm must be sufficiently basic and in practice they can be carries out easily.

An algorithm must possess the following properties

1. Finiteness: An algorithm must terminate in a finite number of steps

2. Definiteness: Each step of the algorithm must be precisely and unambiguously stated

3. Effectiveness: Each step must be effective, in the sense that it should be primitive easily convert able into program statement) can be performed exactly in a finite amount of time.

4. Generality: The algorithm must be complete in itself so that it can be used to solve problems of a specific type for any input data.

5. Input/output: Each algorithm must take zero, one or more quantities as input data produce one or more output values. An algorithm can be written in English like sentences or in any standard representation sometimes, algorithm written in English like languages are called Pseudo Code

Example

1. 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

1.2.2 Algorithms for Simple Problem

Write an algorithm for the following

1. Write an algorithm to calculate the simple interest using the formula.

   Simple interest = P*N* R/100.

Where P is principle Amount, N is the number of years and R is the rate of interest.

Step 1: Read the three input quantities' P, N and R.

Step 2 : Calculate simple interest as

Simple interest = P* N* R/100

Step 3: Print simple interest.

Step 4: Stop.

2. **Area of Triangle**: 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.

3. Write an algorithm to find the largest of three numbers X, Y,Z.

Step 1: Read the numbers X,Y,Z.

Step 2: if (X > Y)

Big = X

else BIG = Y

Step 3 : if (BIG < Z)

Step 4: Big = Z

Step 5: Print the largest number i.e. Big

Step 6: Stop.

4. Write down an algorithm to find the largest data value of a set of given data values

Algorithm largest of all data values:

Step 1: LARGE ← 0

Step 2: read NUM

Step 3: While NUM > = 0 do

3.1 if NUM > LARGE

3.1.1 then

3.1.1.1 LARGE ← NUM

3.2. read NUM

Step 4: Write "largest data value is", LARGE

Step 5: end.

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

Algorithm prime testing:

Step 1: M ← 2

Step 2: read N

Step 3: MAX ← SQRT (N)

Step 4: While M < = MAX do

4.1 if (M* (N/M) = N

4.1.1 then

4.1.1.1 go to step 7

4.2. M ← M + 1

Step 5: Write "number is prime"

Step 6: go to step 8

Step 7: Write "number is not a prime"

Step 8: end.

6. Write algorithm to find the factorial of a given number N

Step 1: PROD ← 1

Step 2: I ← 0

Step 3: read N

Step 4: While I < N do

4.1 I ← I + 1

4.2. PROD ← PROD* I

Step 5: Write "Factorial of", N, "is", PROD

Step 6: end.

7. Write an algorithm to find sum of given data values until negative value is entered.

Algorithm Find – Sum

Step 1: SUM ← 0

Step 2: I ← 0

Step 3: read NEW VALUE

Step 4: While NEW VALUE < = 0 do

4.1 SUM ← SUM + NEW VALUE

4.2 1 ← I + 1

4.3 read NEW VALUE

Step 5: Write "Sum of", I, "data value is, "SUM

Step 6: END

8. Write an algorithm to calculate the perimeter and area of rectangle. Given its length and width.

Step 1: Read length of the rectangle.

Step 2: Read width of the rectangle.

Step 3: Calculate perimeter of the rectangle using the formula perimeter = 2* (length + width)

Step 4: Calculate area of the rectangle using the formula area = length *width.

Step 5: Print perimeter.

Step 6: Print area.

Step 7: Stop.

1.3 Flowchart

A flow chart is a step by step diagrammatic representation of the logic paths to solve a given problem. Or A flowchart is visual or graphical representation of an algorithm.

The flowcharts are pictorial representation of the methods to b used to solve a given problem and help a great deal to analyze the problem and plan its solution in a systematic and orderly manner. A flowchart when translated in to a proper computer language, results in a complete program.

Advantages of Flowcharts

1. The flowchart shows the logic of a problem displayed in pictorial fashion 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.

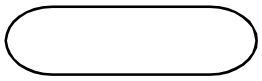
Differences between Algorithm and Flowchart

Algorithm	Flowchart
1. A method of representing the step-by-step logical procedure for solving a problem	1. Flowchart is diagrammatic representation of an algorithm. It is constructed using different types of boxes and symbols.
2. It contains step-by-step English descriptions, each step representing a particular operation leading to solution of problem	2. The flowchart employs a series of blocks and arrows, each of which represents a particular step in an algorithm
3. These are particularly useful for small problems	3. These are useful for detailed representations of complicated programs
4. For complex programs, algorithms prove to be Inadequate	4. For complex programs, Flowcharts prove to be adequate

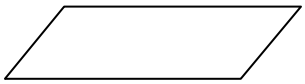
1.4 Symbols used in Flow-Charts

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

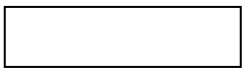
a. Oval: Rectangle with rounded sides is used to indicate either START/ STOP of the program. ..



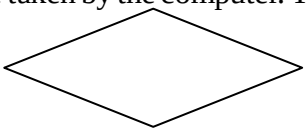
b. Input and output indicators: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are represented in these Parallelograms.



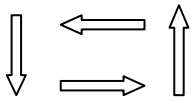
c. Process Indicators: - Rectangle is used to indicate any set of processing operation such as for storing arithmetic operations.



d. 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

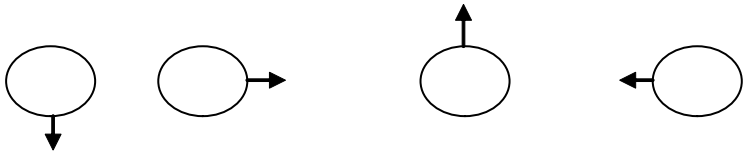


e. Flow Lines: Flow lines indicate the direction being followed in the flowchart. In a Flowchart, every line must have an arrow on it to indicate the direction. The arrows may be in any direction

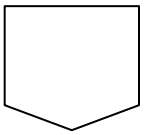


f. On- Page connectors: Circles are used to join the different parts of a flowchart and these circles are called on-page connectors. The uses of these connectors give a neat shape to the flowcharts. Ina complicated problems, a flowchart may run in to several pages. The parts of the flowchart on different

pages are to be joined with each other. The parts to be joined are indicated by the circle.



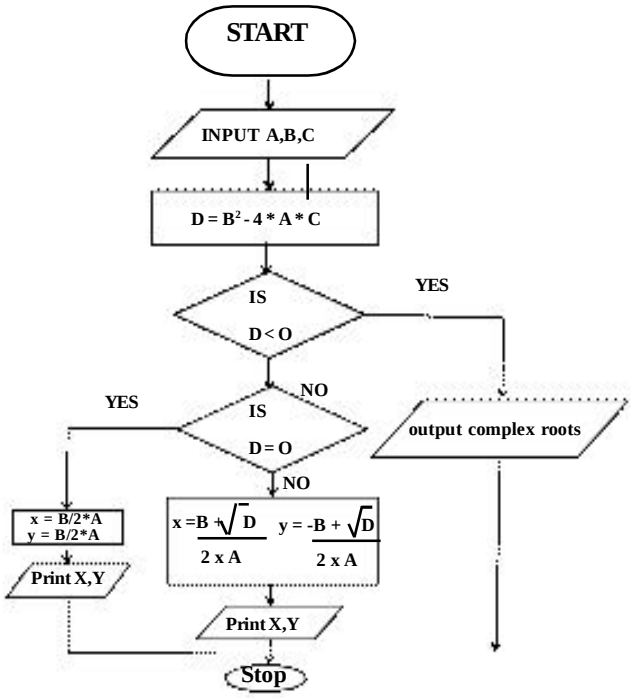
g. Off-page connectors: This connector represents a break in the path of flowchart which is too large to fit on a single page. It is similar to on-page connector. The connector symbol marks where the algorithm ends on the first page and where it continues on the second.



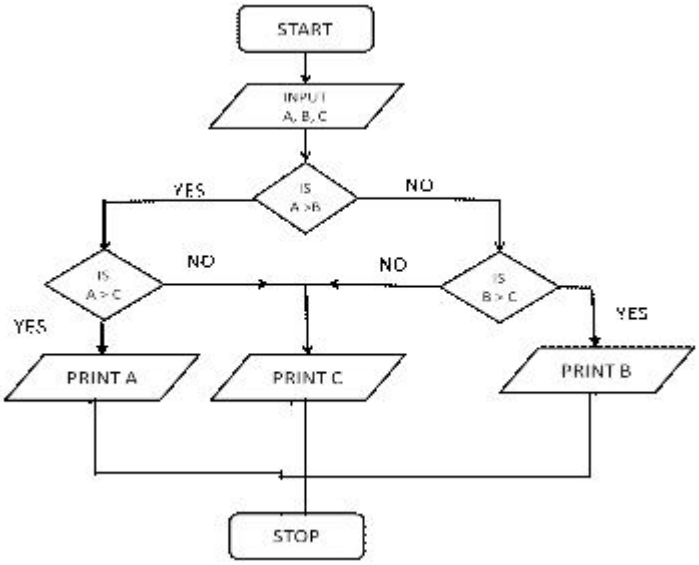
1.4.1 Simple Problems using Flow Chart

Draw the Flowchart for the following

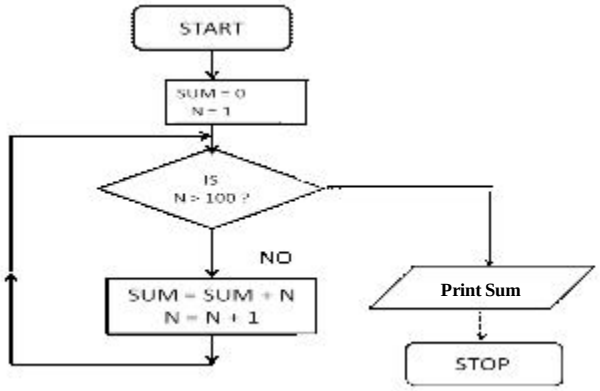
1. Draw the Flowchart to find Roots of Quadratic equation $ax^2 + bx + c = 0$. The coefficients a, b, c are the input data



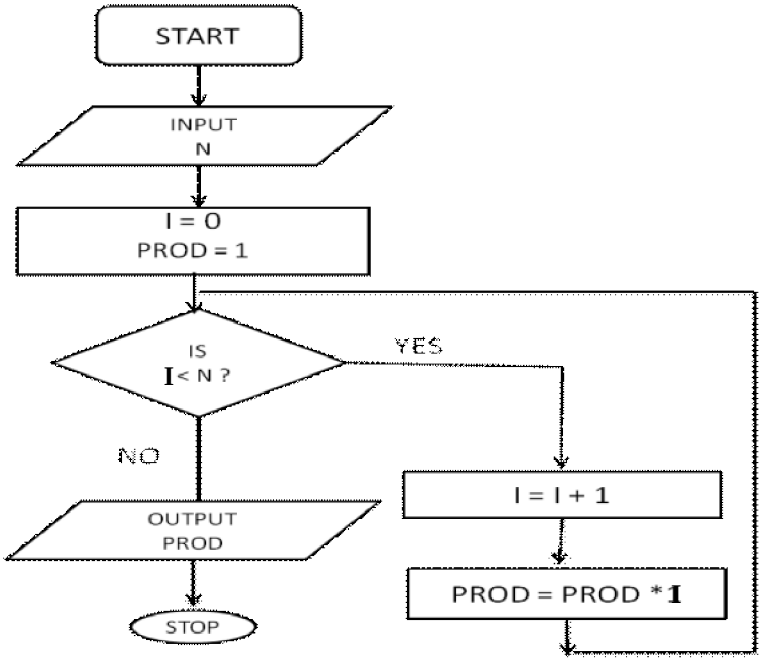
2. Draw a flowchart to find out the biggest of the three unequal positive numbers.



3. Draw a flowchart for adding the integers from 1 to 100 and to print the sum.

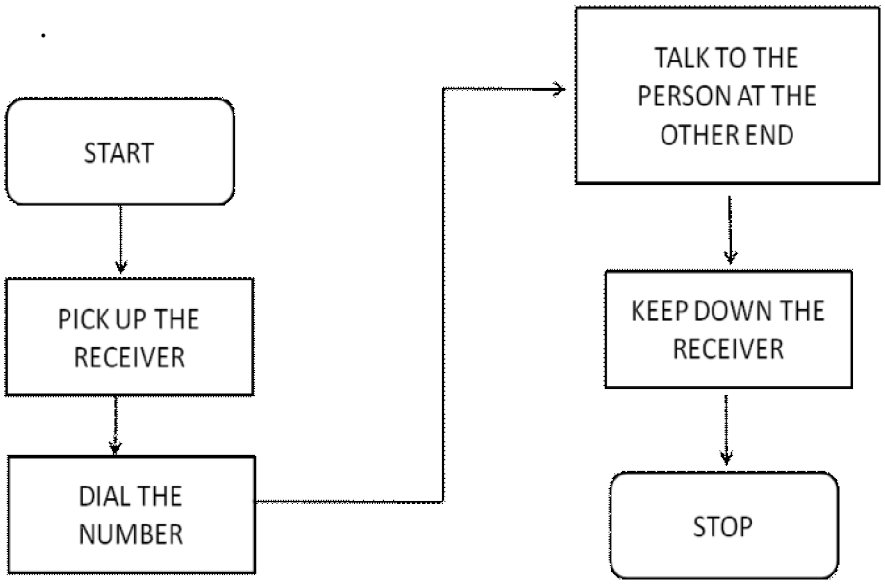


4. Draw a flowchart to find the factorial of given positive integer N.



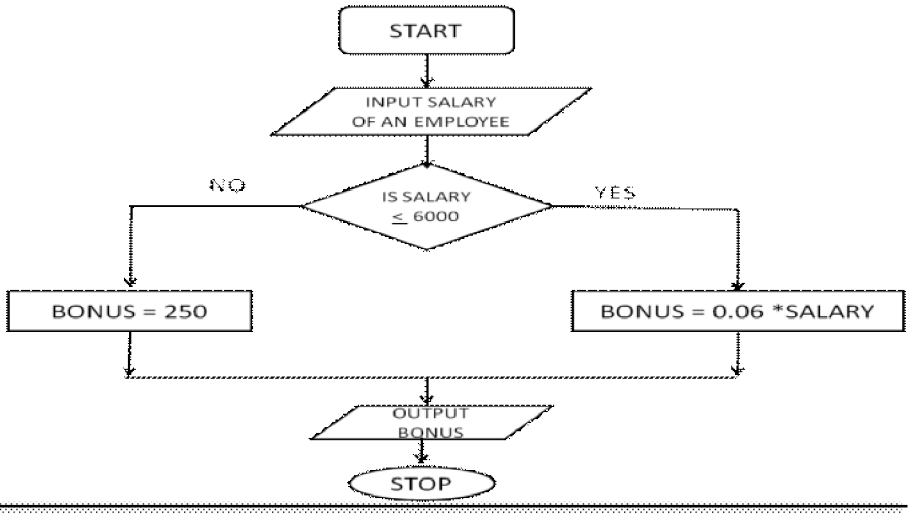
5. Develop a flowchart to illustrate how to make a Land phone telephone call

.



Flowchart for Telephone call

6. 6. ABC company plans to give a 6% year-end bonus to each of its employees earning Rs 6,000 or more per month , and a fixed Rs 250/- - bonus to the remaining employees. Draw a flowchart for calculating the bonus for an employee



1.5 Pseudo code

The Pseudo code is neither an algorithm nor a program. It is an abstract form of a program. It consists of English like statements which perform the specific operations. It is defined for an algorithm. It does not use any graphical representation. In pseudo code, the program is represented in terms of words and phrases, but the syntax of program is not strictly followed.

Advantages: * Easy to read, * Easy to understand, * Easy to modify.

Example*:* Write a pseudo code to perform the basic arithmetic operations.

Read n1, n2

Sum = n1 + n2

Diff = n1 – n2

Mult = n1 * n2

Quot = n1/n2

Print sum, diff, mult, quot

End.

Activity

Practice more sample problems on algorithm and Flowcharts

Model Questions

Short Answer Type Questions - 2 Marks

1. Define Algorithm
2. What is Flowchart
3. What is Pseudo code?
4. What are the symbols of Flowchart
5. Write an Algorithm for perimeter of Triangle
6. What are the basic steps involved In problem solving

Long Answer Type Questions - 6 Marks

1. Differentiate between Algorithm and Flowchart.
2. Write an algorithm to find greatest of given three numbers.
3. Write an algorithm to check whether given integer value is PRIME or NOT.
4. Draw the flowchart to find roots of Quadratic equation $ax^2 + bx + c = 0$

Note : Practice more related Algorithms and Flowcharts.

UNIT 2

Features of 'C'

Structure

Learning Objectives

- What is C Language and its importance
- To understand various data types
- To understand working function of input and output statements in C
- To understand working function of Branching statements in C
- To understand working function of Looping statements in C
- To Understand differences between Break and Continue

2.0 Introduction

'C' is high level language and is the upgraded version of another language (Basic Combined Program Language). C language was designed at Bell laboratories in the early 1970's by Dennis Ritchie. C being popular in the modern computer world can be used in Mathematical Scientific, Engineering and Commercial applications

The most popular Operating system UNIX is written in C language. This language also has the features of low level languages and hence called as "System Programming Language"

Features of C language

- Simple, versatile, general purpose language
- It has rich set of Operators
- Program execution are fast and efficient
- Can easily manipulates with bits, bytes and addresses
- Varieties of data types are available
- Separate compilation of functions is possible and such functions can be called by any C program
- Block- structured language
- Can be applied in System programming areas like operating systems, compilers & Interpreters, Assembles, Text Editors, Print Spoolers, Network Drivers, Modern Programs, Data Bases, Language Interpreters, Utilities etc.

2.1 Character Set

The character set is the fundamental raw-material for any language. Like natural languages, computer languages will also have well defined character-set, which is useful to build the programs.

The C language consists of two character sets namely – source character set execution character set. Source character set is useful to construct the statements in the source program. Execution character set is employed at the time of execution of h program.

1. Source character set : This type of character set includes three types of characters namely alphabets, Decimals and special symbols.

i. Alphabets : A to Z, a to z and Underscore(_)

ii. Decimal digits : 0 to 9

iii. Special symbols: + - * / ^ % = & ! ( ) { } [ ] " etc

2. Execution character set : This set of characters are also called as non-graphic characters because these are invisible and cannot be printed or displayed directly.

These characters will have effect only when the program being executed. These characters are represented by a back slash (\) followed by a character.

Execution character	Meaning	Result at the time of execution
\ n	End of a line	Transfers the active position of cursor to the initial position of next line
\ 0 (zero)	End of string	Null
\ t	Horizontal Tab	Transfers the active position of cursor to the next Horizontal Tab
\ v	Vertical Tab	Transfers the active position of cursor to the next Vertical Tab
\ f	Form feed	Transfers the active position of cursor to the next logical page
\ r	Carriage return	Transfers the active position of cursor to the initial position of current line

2.2 Structure of a 'C' Program

The Complete structure of C program is

The basic components of a C program are:

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

Preprocessor Statements: These statements begin with # symbol. They are called preprocessor directives. These statements direct the C preprocessor to include header files and also symbolic constants in to C program. Some of the preprocessor statements are

#include<stdio.h>: for the standard input/output functions

#include<test.h>: for file inclusion of header file Test.

#define NULL 0: for defining symbolic constant NULL = 0 etc.

Global Declarations: Variables or functions whose existence is known in the main() function and other user defined functions are called global variables (or functions) and their declarations is called global declaration. This declaration should be made before main().

main(): As the name itself indicates it is the main function of every C program. Execution of C program starts from main (). No C program is executed without main() function. It should be written in lowercase letters and should not be terminated by a semicolon. It calls other Library functions user defined functions. There must be one and only one main() function in every C program.

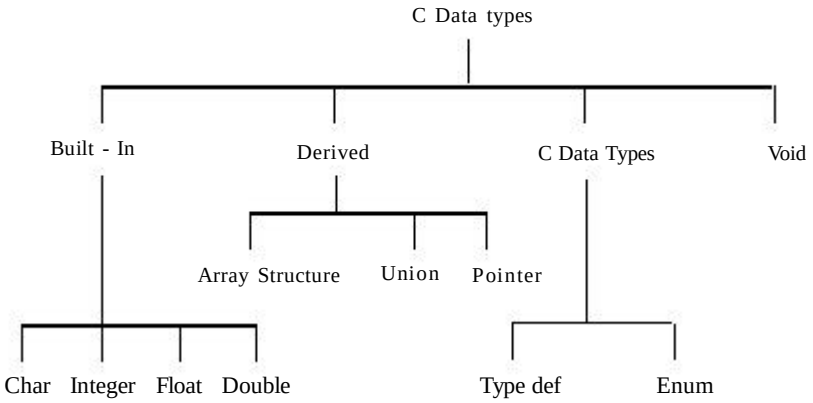
Braces: Every C program uses a pair of curly braces ({,}0. The left brace indicates beginning of main() function. On the other hand, the right brace indicates end of the main() function. The braces can also be used to indicate the beginning and end of user-defined functions and compound statements.

Declarations: It is part of 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 specific operations. They may be input-output statements, arithmetic statements, control statements and other statements. They are also including comments.

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

2.3 Data Types in 'C'



The built-in data types and their extensions is the subject of this chapter. Derived data types such as arrays, structures, union and pointers and user defined data types such as typedef and enum.

Basic Data Types

There are four basic data types in C language. They are Integer data, character data, floating point data and double data types.

a. Character data: Any character of the ASCII character set can be considered as a character data types and its maximum size can be 1 byte or 8 byte long. 'Char' is the keyword used to represent character data type in C.

Char - a single byte size, capable of holding one character.

b. Integer data: The keyword 'int' stands for the integer data type in C and its size is either 16 or 32 bits. The integer data type can again be classified as

1. Long int - long integer with more digits
2. Short int - short integer with fewer digits.
3. Unsigned int - Unsigned integer

4. Unsigned short int – Unsigned short integer

5. Unsigned long int – Unsigned long integer

As above, the qualifiers like short, long, signed or unsigned can be applied to basic data types to derive new data types.

int - an Integer with the natural size of the host machine.

c. Floating point data: - The numbers which are stored in floating point representation with mantissa and exponent are called floating point (real) numbers. These numbers can be declared as 'float' in C.

float – Single – precision floating point number value.

d. Double data : - Double is a keyword in C to represent double precision floating point numbers.

double - Double – precision floating point number value.

Data Kinds in C

Various data kinds that can be included in any C program can fall in to the following.

a. Constants/Literals

b. Reserve Words Keywords

c. Delimeters

d. Variables/Identifiers

a. Constans/Literals: Constants are those, which do not change, during the execution of the program. Constants may be categorized in to:

- Numeric Constants
- Character Constants
- String Constants

1. Numeric Constants

Numeric constants, as the name itself indicates, are those which consist of numerals, an optional sign and an optional period. They are further divided into two types:

(a) Integer Constants (b) Real Constants

a. Integer Constants

A whole number is an integer constant Integer constants do not have a decimal point. These are further divided into three types depending on the number systems they belong to. They are:

i. Decimal integer constants

ii. Octal integer constants

iii. Hexadecimal integer constants

i. A decimal integer constant is characterized by the following properties

- It is a sequence of one or more digits ([0…9], the symbols of decimal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid decimal integer constants:

456

-123

Some examples of invalid decimal integer constants:

4.56 - Decimal point is not permissible

1,23 - Commas are not permitted

ii. An octal integer constant is characterized by the following properties

- It is a sequence of one or more digits ([0…7], symbols of octal number system).
- It may have an optional + or – sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the digit 0.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid octal integer constants:

0456

-0123

+0123

Some examples of invalid octal integer constants:

04.56 - Decimal point is not permissible

04,56 - Commas are not permitted

x34 - x is not permissible symbol

568 - 8 is not a permissible symbol

iii. An hexadecimal integer constant is characterized by the following properties

- It is a sequence of one or more symbols ([0…9][A….Z], the symbols of Hexadecimal number system).
- It may have an optional + or - sign. In the absence of sign, the constant is assumed to be positive.
- It should start with the symbols 0X or 0x.
- Commas and blank spaces are not permitted.
- It should not have a period as part of it.

Some examples of valid hexadecimal integer constants:

0x456

-0x123

0x56A

0XB78

Some examples of invalid hexadecimal integer constants:

0x4.56 - Decimal point is not permissible

0x4,56 - Commas are not permitted.

b. Real Constants

The real constants also known as *floating point constants* are written in two forms:

(i) Fractional form, (ii) Exponential form.

i. Fractional Form

The real constants in Fractional form are characterized by the following characteristics:

- Must have at least one digit.
- Must have a decimal point.
- May be positive or negative and in the absence of sign taken as positive.
- Must not contain blanks or commas in between digits.
- May be represented in exponential form, if the value is too higher or too low.

Some examples of valid real constants:

456.78

-123.56

Some examples of invalid real constants:

4.56 - Blank spaces are not permitted

4,56 - Commas are not permitted

456 - Decimal point missing

ii. Exponential Form

The exponential form offers a convenient way for writing very large and small real constant. For example, 56000000.00, which can be written as 0.56 *, 10^8 is written as *0.56E8* or *0.56e8* in exponential form. 0.000000234, which can be written as 0.234 * 10^{-6} is written as *0.234E-6* or *0.234e-6* in exponential form. The letter E or e stand for exponential form.

A real constant expressed in exponential form has two parts: (i) Mantissa part, (ii) Exponent part. Mantissa is the part of the real constant to the left of *E* or *e*, and the Exponent of a real constant is to the right of *E* or *e*. Mantissa and Exponent of the above two number are shown below.

Mantissa ↓	Exponent ↓	Mantissa ↓	Exponent ↓
0.56	E 8	0.234	E -6

In the above examples, 0.56 and 0.234 are the mantissa parts of the first and second numbers, respectively, and 8 and -6 are the exponent parts of the first and second number, respectively.

The real constants in exponential form and characterized by the following characteristics:

- The mantissa must have at least one digit.
- The mantissa is followed by the letter E or e and the exponent.
- The exponent must have at least one digit and must be an integer.
- A sign for the exponent is optional.

Some examples of valid real constants:

3E4

23e-6

0.34E6

Some examples of invalid real constants:

23E - No digit specified for exponent

23e4.5 - Exponent should not be a fraction

23,4e5 - Commas are not allowed

256*e8- * not allowed

2. Character Constants

Any character enclosed with in single quotes (‘) is called character constant. A character constant:

- May be a single alphabet, single digit or single special character placed with in single quotes.
- Has a maximum length of 1 character.

Here are some examples,

- ‘C’
- ‘c’
- ‘:’
- ‘*’

3. String Constants

A string constant is a sequence of alphanumeric characters enclosed in double quotes whose maximum length is 255 characters.

Following are the examples of valid string constants:

- "My name is Krishna"
- "Bible"
- "Salary is 18000.00"

Following are the examples of invalid string constants:

My name is Krishna - Character are not enclosed in double quotation marks.

"My name is Krishna - Closing double quotation mark is missing.

'My name is Krishna' - Characters are not enclosed in double quotation marks

b. Reserve Words/Keywords

In C language , some words are reserved to do specific tasks intended for them and are called Keywords or Reserve words. The list reserve words are

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

c. Delimiters

This is symbol that has syntactic meaning and has got significance. These will not specify any operation to result in a value. C language delimiters list is given below

Symbol	Name	Meaning
#	Hash	Pre-processor directive
,	comma	Variable delimiter to separate variable
:	colon	label delimiter
;	Semicolon	statement delimiter
( )	parenthesis	used for expressions
{ }	curly braces	used for blocking of statements
[ ]	square braces	used along with arrays

d. Variables / Identifiers

These are the names of the objects, whose values can be changed during the program execution. Variables are named with description that transmits the value it holds.

[A quantity of an item, which can be change its value during the execution of program is called variable. It is also known as Identifier].

Rules for naming a variable:-

- It can be of letters, digits and underscore(_)
- First letter should be a letter or an underscore, but it should not be a digit.
- Reserve words cannot be used as variable names**.**

Example: basic, root, rate, roll-no etc are valid names.

Declaration of variables:

Syntax	type	Variable list
int	i, j	i, j are declared as integers
float	salary	salary is declared ad floating point variable
Char	sex	sex is declared as character variable

2.4 Operators

An Operator is a symbol that operates on a certain data type. The data items that operators act upon are called **operands**. Some operators require two operands, some operators act upon only one operand. In C, operators can be classified into various categories based on their utility and action.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operator
4. Assignment Operator
5. Increment & Decrement Operator
6. Conditional Operator
7. Bitwise Operator
8. Comma Operator

1. Arithmetic Operators

The Arithmetic operators performs arithmetic operations. The Arithmetic operators can operate on any built in data type. A list of arithmetic operators are

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division

2. Relational Operators

Relational Operators are used to compare arithmetic, logical and character expressions. The Relational Operators compare their left hand side expression with their right hand side expression. Then evaluates to an integer. If the Expression is false it evaluate to “zero”(0) if the expression is true it evaluate to “one”

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greater than or Equal to
= =	Equal to

!=	Not Equal to

The Relational Operators are represented in the following manner:

Expression-1 Relational Operator Expression-2

The Expression-1 will be compared with Expression -2 and depending on the relation the result will be either "TRUE" OR "FALSE".

Examples :

Expression Evaluate to

(5 <= 10) ——————— 1

(-35 > 10) ——————— 0

(X < 10) ——————— 1 (if value of x is less than 10)

0 Other wise

(a + b) = = (c + d) 1 (if sum of a and b is equal to sum of c, d)

0 Other wise

3. Logical Operators

A logical operator is used to evaluate logical and relational expressions. The logical operators act upon operands that are themselves logical expressions. There are three logical operators.

Operators	Expression
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical And (&&): A compound Expression is true when two expression when two expressions are true. The && is used in the following manner.

Exp1 && Exp2.

The result of a logical AND operation will be true only if both operands are true.

The results of logical operators are:

Exp1 Op. Exp2 Result

True && True True

True && False False

False && False False

False && True False

Example: a = 5; b = 10; c = 15;

Exp1 Exp2 Result

1. (a< b) && (b < c) => True

2. (a> b) && (b < c) => False

3. (a< b) && (b > c) => False

4. (a> b) && (b > c) => False

Logical OR*:* A compound expression is false when all expression are false otherwise the compound expression is true. The operator "||" is used as It evaluates to true if either exp-1 or exp-2 is true. The truth table of "OR" is Exp1 || Exp2

Exp1 Operator Exp2 Result:

True || True True

True || False True

False || True True

False || False False

Example: a = 5; b = 10; c = 15;

Exp1 Exp2 Result

1. (a< b) || (b < c) => True

2. (a> b) || (b < c) => True

3. (a< b) || (b > c) => True

4. (a> b) || (b > c) => False

Logical NOT: The **NOT (!)** operator takes single expression and evaluates to true(1) if the expression is false (0) or it evaluates to false (0) if expression is true (1). The general form of the expression.

! (Relational Expression)

The truth table of **NOT :**

Operator. Exp1 Result

! True False

! False True

Example: a = 5; b = 10; c = 15

1. !(a< b) False

2. !(a> b) True

4. Assignment Operator

An assignment operator is used to assign a value to a variable. The most commonly used assignment operator is **=**. The general format for assignment operator is :

<Identifer> = < expression >

Where identifier represent a variable and expression represents a constant, a variable or a Complex expression.

If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left.

Example: Suppose that **I** is an **Integer** type Variable then

1. I = 3.3 3 (Value of I)

2. I = 3.9 3 (Value of I)

3. I = 5.74 5 (Value of I)

Multiple assignment

< identifier-1 > = < identifier-2 > = - - - = < identifier-n > = <exp>;

Example: a,b,c are integers; j is float variable

1. a = b = c = 3;

2. a = j = 5.6; then a = 5 and j value will be 5.6

C contains the following five additional assignment operators

1. += **2.**-= **3.** += **4.** *= **5.** /=

The assignment expression is: - Exp1 < Operator> Exp-2

Ex: I = 10 (assume that)

Expression Equivalent to Final Value of 'I'

1. I + = 5 I = I + 5 15

2. I - = 5 I = I - 5 10

3. I * = 5 I = I * 5 50

4. I / = 5 I = I / 5 10

5. Increment & Decrement Operator

The increment/decrement operator act upon a Single operand and produce a new value is also called as "**unary operator"**. The increment operator ++ adds 1 to the operand and the Decrement operator – subtracts 1 from the operand.

Syntax: < operator >< variable name >;

The ++ or – operator can be used in the two ways.

Example : ++ a; Pre-increment (or) a++ Post increment —a; Pre-Decrement (or) a— Post decrement

1. ++ a Immediately increments the value of a by 1.

2. a ++ The value of the a will be increment by 1 after it is utilized.

Example 1*:* Suppose a = 5 ;

Statements Output

printf ("a value is %d", a); a value is 5

printf ("a value is %d", ++ a); a value is 6

printf ("a value is %d ", a) ; a value is 6

Example 2*:* Suppose : a = 5 ;

Statements Output

printf ("a value is %d ", a); a value is 5

printf ("a value is %d ", a++); a value is 5

printf ("a value is %d ",a); a value is 6

a and a- will be act on operand by decrement value like increment operator.

6. Conditional operator (or) Ternary operator (? :)

It is called ternary because it uses three expression. The ternary operator acts like If- Else construction.

Syn :(<Exp –1 > ? <Exp-2> : <Exp-3>);

Expression-1 is evaluated first. If Exp-1 is true then Exp-2 is evaluated other wise it evaluate Exp-3 will be evaluated.

Flow Chart :

Exp-1

Exp-2 Exp-3

Exit

Example:

1. a = 5 ; b = 3;

(a> b ? printf (“a is larger”) : printf (“b is larger”));

Output is :a is larger

2. a = 3; b = 3;

(a> b ? printf (“a is larger”) : printf (“b is larger”));

Output is :b is larger

7. Bit wise Operator

A bitwise operator operates on each bit of data. These bitwiseoperator can be divided into three categories.

i. The logical bitwise operators.

ii. The shift operators

iii. The one’s complement operator.

i) The logical Bitwise Operator : There are three logical bitwise operators.

Meaning Operator:

a) Bitwise **AND &**

b) Bitwise **OR |**

c) Bitwise exclusive **XOR ^**

Suppose b1 and b2 represent the corresponding bits with in the first and second operands, respectively.

B1 B2 B1 & B2 B1 | B2 B1 ^ B2

1 1 1 1 0

1 0 0 1 1

0 1 0 1 1

0 0 0 0 0

The operations are carried out independently on each pair of corresponding bits within the operand thus the least significant bits (ie the right most bits) within the two operands. Will be compared until all the bits have been compared. The results of these comparisons are

A **Bitwise AND** expression will return a 1 if both bits have a value of 1. Other wise, it will return a value of 0.

A **Bitwise OR** expression will return a 1 if one or more of the bits have a value of 1. Otherwise, it will return a value of 0.

A **Bitwise EXCLUSIVE OR** expression will return a 1 if one of the bits has a value of 1 and the other has a value of 0. Otherwise, if will return a value of 0.

Example::**Variable Value Binary Pattern**

X 5 0101

Y 2 0010

X & Y 0 0000

X | Y 7 0111

X ^ Y 7 0111

ii) The Bitwise shift Operations: The two bitwise shift operators are **Shift left** (<<) and **Shift right** (>>). Each operator requires two operands. The first operand that represents the bit pattern to be shifted. The second is an unsigned integer that indicates the number of displacements.

Example: c = a << 3;

The value in the integer a is shifted to the left by three bit position. The result is assigned to the c.

A = 13; c= A<<3;

Left shit << c= 13 * 2 3 = 104;

Binary no 0000 0000 0000 1101

After left bit shift by 3 places ie,. a<<3

0000 0000 0110 1000

The right –bit – shift operator (>>) is also a binary operator.

Example: c = a >> 2 ;

The value of a is shifted to the right by 2 position

insert 0's Right – shift >> drop off 0's

0000 0000 0000 1101

After right shift by 2 places is a>>2

0000 0000 0000 0011 c=13>>2

c= 13/4=3

iii) Bit wise complement: The complement op.~ switches all the bits in a binary pattern, that is all the 0's becomes 1's and all the 1's becomes 0's.

variable value Binary patter

x 23 0001 0111

~x 132 1110 1000

8. Comma Operator

A set of expressions separated by using commas is a valid construction in c language.

Example : int i, j;

i= (j = 3, j + 2) ;

The first expression is j = 3 and second is j + 2. These expressions are evaluated from left to right. From the above example I = 5.

Size of operator: The operator size operator gives the size of the data type or variable in terms of bytes occupied in the memory. This operator allows a determination of the no of bytes allocated to various Data items

Example : int i; float x; double d; char c; **OUTPUT**

Printf ("integer : %d\n", sizeof(i)); Integer : 2

Printf ("float : %d\n", sizeof(i)); Float : 4

Printf ("double : %d\n", sizeof(i)); double : 8

Printf ("char : %d\n",sizeof(i)); character : 1

2.5 Expressions

An expression can be defined as collection of data object and operators that can be evaluated to lead a single new data object. A data object is a constant, variable or another data object.

Example : a + b

x + y + 6.0

3.14 * r * r

(a + b) * (a – b)

The above expressions are called as arithmetic expressions because the data objects (constants and variables) are connected using arithmetic operators.

Evaluation Procedure: The evaluation of arithmetic expressions is as per the hierarchy rules governed by the C compiler. The precedence or hierarchy rules for arithmetic expressions are

1. The expression is scanned from left to right.

2. While scanning the expression, the evaluation preference for the operators are

*, /, % - evaluated first

+, - - evaluated next

3. To overcome the above precedence rules, user has to make use of parenthesis. If parenthesis is used, the expression/ expressions with in parenthesis are evaluated first as per the above hierarchy.

Statements

Data Input & Output

An input/output function can be accessed from anywhere within a program simply by writing the function name followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function.

Some input/output Functions do not require arguments though the empty parentheses must still appear. They are:

	Input Statements	*Output Statements*
Formatted	scanf()	printf()
Unformatted	getchar()gets()	putchar()puts()

getchar()

Single characters can be entered into the computer using the C library Function **getchar()**. It returns a single character from a standard input device. The function does not require any arguments.

Syntax: <Character variable> = getchar();

Example: char c;

c = getchar();

putchar()

Single characters can be displayed using function **putchar()**. It returns a single character to a standard output device. It must be expressed as an argument to the function.

Syntax: putchar(<character variable>);

Example: char c;

_____

putchar(c);

gets()

The function **gets()** receives the string from the standard input device.

Syntax: gets(<string type variable or array of char>);

Where s is a string.

The function gets accepts the string as a parameter from the keyboard, till a newline character is encountered. At end the function appends a "null" terminator and returns.

puts()

The function **puts()** outputs the string to the standard output device.

Syntax: puts(s);

Where s is a string that was real with gets();

Example:

```
main()
{
char line[80];
gets(line);
puts(line);
}
```

scanf()

Scanf() function can be used input the data into the memory from the standard input device. This function can be used to enter any combination of numerical Values, single characters and strings. The function returns number of data items.

Syntax:-scanf ("control strings", &arg1,&arg2,——&argn);

Where control string referes to a string containing certain required formatting information and arg1, arg2——argn are arguments that represent the individual input data items.

Example*:*

```
#include<stdio.h>
main()
{
char item[20];
intpartno;
float cost;
scanf("%s %d %f",&item,&partno,&cost);
}
```

Where s, d, f with % are conversion characters. The conversion characters indicate the type of the corresponding data. Commonly used conversion characters from data input.

Conversion Characters

Characters	Meaning
%c	data item is a single character.
%d	data item is a decimal integer.
%f	data item is a floating point value.
%e	data item is a floating point value.
%g	data item is a floating point value.
%h	data item is a short integer.
%s	data item is a string.
%x	data item is a hexadecimal integer.
%o	data item is a octal interger.

printf()

The printf() function is used to print the data from the computer's memory onto a standard output device. This function can be used to output any combination of numerical values, single character and strings.

Syntax: printf("control string", arg-1, arg-2,————arg-n);

Where control string is a string that contains formatted information, and arg-1, arg-2 —— are arguments that represent the output data items.

Example:

```
#include<stdio.h>
main()
{
char item[20];
intpartno;
float cost;
_____
printf ("%s %d %f", item, partno, cost);
}                    (Where %s %d %f are conversion characters.)
```

2.6 Assignment Statement

Assignment statement can be defined as the statement through which the value obtained from an expression can be stored in a variable.

The general form of assignment statement is

< variable name> = < arithmetic expression> ;

Example: sum = a + b + c;

tot = s1 + s2 + s3;

area = ½ * b* h;

2.7 I/O Control Structure (if, If-else, for, while, do-while)

Conditional Statements

The conditional expressions are mainly used for decision making. The following statements are used to perform the task of the conditional operations.

a. if statement.

b. If-else statement. Or 2 way if statement

c. Nested else-if statement.

d. Nested if –else statement.

e. Switch statement.

a. if statement

The **if statement** is used to express conditional expressions. If the given condition is true then it will execute the statements otherwise skip the statements.

The simple structure of '**if**' statement is

i. If (< condtional expressione>)

statement-1;

(or)

ii. If (< condtional expressione>)

{

```
statement-1;
statement-2;
statement-3;
……
……
STATEMENT-N
}
```

The expression is evaluated and if the expression is true the statements will be executed. If the expression is false the statements are skipped and execution continues with the next statements.

Example: a=20; b=10;

```
if ( a > b )
printf ("big number is %d" a);
```

b. if-else statements

The **if-else** statements is used to execute the either of the two statements depending upon the value of the exp. The general form is

```
if(<exp>)
{
Statement-1;
Statement -2;
……..          " SET-I"
……
Statement- n;
}
else
{
Statement1;
Statement 2;
```

…….. " SET-II

……

Statement n;

}

SET - I Statements will be executed if the exp is true.

SET – II Statements will be executed if the exp is false.

Example:

if (a> b)

printf ("a is greater than b");

e**lse**

printf ("a is not greater than b");

c. Nested else-if statements

If some situations if may be desired to nest multiple **if-else** statements. In this situation one of several different course of action will be selected.

Syntax

if (<exp1>)

Statement-1;

else if (<exp2>)

Statement-2;

else if (<exp3>)

Statement-3;

else

Statement-4;

When a logical expression is encountered whose value is true the corresponding statements will be executed and the remainder of the nested else if statement will be bypassed. Thus control will be transferred out of the entire nest once a true condition is encountered.

The final **else** clause will be apply if none of the exp is true.

d. nestedif-else statement

It is possible to nest if-else statements, one within another. There are several different form that nested if-else statements can take.

The most general form of two-layer nesting is

```
if(exp1)
        if(exp3)
    Statement-3;
        else
    Statement-4;
else
        if(exp2)
    Statement-1;
        else
    Statement-2;
```

One complete **if-else** statement will be executed if **expression1** is true and another complete **if-else** statement will be executed if **expression1** is false.

e. Switch statement

A switch statement is used to choose a statement (for a group of statement) among several alternatives. The switch statements is useful when a variable is to be compared with different constants and in case it is equal to a constant a set of statements are to be executed.

Syntax:

```
Switch (exp)
{
case
    constant-1:
    statements1;
case
    constant-2:
```

```
        statements2;
        _______
        _______
    default:
        statement n;
    }
```

Where constant1, constanat2 — — — are either integer constants or character constants. When the switch statement is executed the exp is evaluated and control is transferred directly to the group of statement whose case label value matches the value of the exp. If none of the case label values matches to the value of the exp then the default part statements will be executed.

If none of the case labels matches to the value of the exp and the default group is not present then no action will be taken by the switch statement and control will be transferred out of the switch statement.

A simple switch statement is illustrated below.

Example 1:

```
main()
{
char choice;
printf("Enter Your Color (Red - R/r, White – W/w)");
choice=getchar();
switch(choice= getchar())
{
case'r':
case'R':
printf ("Red");
break;
case'w':
case'W':
```

```
printf ("white");
break;
default :
printf ("no colour");
}
```

Example 2:

```
switch(day)
{
case 1:
printf ("Monday");
break;
_____
_____
}
```

2.8 Structure for Looping Statements

Loop statements are used to execute the statements repeatedly as long as an expression is true. When the expression becomes false then the control transferred out of the loop. There are three kinds of loops in **C**.

a) while b) do-while c) for

a. while statement

while loop will be executed as long as the exp is true.

Syntax:

```
while (exp)
{
statements;
}
```

The statements will be executed repeatedly as long as the exp is true. If the exp is false then the control is transferred out of the while loop.

Example:

int digit = 1;

While (digit <=5) FALSE

{

printf ("%d", digit); TRUE

Cond Exp

Statements; ++digit;

}

The while loop is top tested i.e., it evaluates the condition before executing statements in the body. Then it is called entry control loop.

b. do-while statement

The **do-while** loop evaluates the condition after the execution of the statements in the body.

Syntax: do

Statement;

While<exp>;

Here also the statements will be executed as long as the exp value is true. If the expression is false the control come out of the loop.

Example:

-int d=1;

do

{

printf ("%d", d); FALSE

++d;

} **while** (d<=5); TRUE

Cond Exp

statements

exit

The statement with in the do-while loop will be executed at least once. So the **do-while** loop is called a bottom tested loop.

c. for statement

The **for** loop is used to executing the structure number of times. The **for** loop includes three expressions. First expression specifies an initial value for an index (initial value), second expression that determines whether or not the loop is continued (conditional statement) and a third expression used to modify the index (increment or decrement) of each pass.

Note: Generally for loop used when the number of passes is known in advance.

Syntax: **for** (exp1;exp2;exp3)

```
{
        Statement –1;
        Statement – 2;
        ————————; FALSE
        ————————;
        Statement - n; TRUE
}
exp2
        Statements;
exp3
Exit loop
exp1
start
```

Where **expression-1** is used to initialize the control variable. This expression is executed this expression is executed is only once at the time of beginning of loop.

Where **expression-2** is a logical expression. If **expression-2** is true, the statements will be executed, other wise the loop will be terminated. This expression is evaluated before every execution of the statement.

Where **expression-3** is an increment or decrement expression after executing the statements, the control is transferred back to the **expression-3** and updated. There are different formats available in **for loop**. Some of the expression of loop can be omit.

Formate - I

```
for( ; exp2; exp3 )
Statements;
```

In this format the initialization expression (i.e., **exp1**) is omitted. The initial value of the variable can be assigned outside of the **for loop.**

Example 1

```
int i = 1;
for( ; i<=10; i++ )
printf ("%d \n", i);
```

Formate - II

```
for( ; exp2 ; )
Statements;
```

In this format the initialization and increment or decrement expression (i.e **expression-1** and **expression-3**) are omitted. The exp-3 can be given at the statement part.

Example 2

```
int i = 1;
for( ; i<=10; )
{
printf ("%d \n",i);
i++;
}
```

Formate - III

```
for( ; ; )
Statements;
```

In this format the **three expressions** are omitted. The loop itself assumes the **expression-2**is true. So **Statements** will be executed infinitely.

Example 3

```
int i = 1;
for ( ; i<=10; )
{
printf ("%d \n",i);
i++;
}
```

2.9 Nested Looping Statements

Many applications require nesting of the loop statements, allowing on loop statement to be embedded with in another loop statement.

Definition

Nesting can be defined as the method of embedding one control structure with in another control structure.

While making control structure s to be reside one with in another ,the inner and outer control structures may be of the same type or may not be of same type. But ,it is essential for us to ensure that one control structure is completely embedded within another.

```
/*program to implement nesting*/
#include <stdio.h>
main()
{
int a,b,c,
for (a=1,a< 2, a++)
{
printf ("%d",a)
for (b=1,b<=2,b++)
{
```

```
print f(%d",b)
for (c=1,c<=2,c++)
{
print f( " My Name is Sunny \n");
                    }
               }
          }
}
```

2.10 Multi Branching Statement (switch), Break, and Continue

For effective handling of the loop structures, C allows the following types of control break statements.

a. Break Statement b. Continue Statement

a. Break Statement

The break statement is used to terminate the control form the loops or to exit from a switch. It can be used within a for, **while, do-while, for**.

The general format is :

```
break;
```

If **break** statement is included in a **while, do-while** or **for** then control will immediately be transferred out of the loop when the break statement is encountered.

Example

```
for ( ; ; ) normal loop
{
break
Condition
within loop
scanf ("%d",&n);
if ( n < -1)
```

```
break;
sum = sum + n;
}
```

b. The Continue Statement

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Rather, the remaining loop statements are skipped and the proceeds directly to the next pass through the loop. The "**continue**" that can be included with in a **while a do-while and a for loop** statement.

General form :

```
continue;
```

The **continue** statement is used for the inverse operation of the **break** statement .

```
Condition
with in loop
Remaining part of loop
continue
```

Example

```
while (x<=100)
{
if (x <= 0)
{
printf ("zero or negative value found \n");
continue;
}
}
```

The above program segment will process only the positive whenever a zero or negative value is encountered, the message will be displayed and it continue the same loop as long as the given condition is satisfied.

2.11 Differences between Break and Continue

Break	Continue
1. Break is a key word used to terminate the loop or exit from the block. The control jumps to next statement after the loop or block	1. Continue is a keyword used for containing the next iteration of the loop
2. Break statements can be used with for, while, do-while, and switch statement. When break is used in nested loops, then only the innermost loop is terminated.	2. This statement when occurs in a loop does not terminate it rather skips the statements after this continue statement and the control goes for next iteration. 'Continue' can be used with for, while and do- while loop.
3. Syntax:{ statement1; statement2; statement3; break;}	3. Syntax: { statement1; continue; statement2; statement3; break; }
4. Example :Switch (choice){ Case 'y': printf("yes"); break; Case 'n': printf("NO"); break;}	4. Example:- I = 1, j=0;While(i<= 7){ I = I + 1; If((I = = 6) Continue; j = j + 1;}
5. When the case matches with the choice entered, the corresponding case block gets executed. When 'break' statement is executed, the control jumps out of the switch statement.	5. In the above loop, when value of ' i becomes 6' continue statement is executed. So, j= j+1 is skipped and control is transferred to beginning of while loop.

2.12 Unconditional Branching (Go To Statement)

goto statement

The **go to statement** is used to alter the program execution sequence by transferring the control to some other part of the program.

Syntax

Where label is an identifier used to label the target statement to which the control would be transferred the target statement will appear as:

Syntax

```
goto<label>;
label :
statements;
```

Example 1

```
#include <stdio.h>
main();
{
inta,b;
printf ("Enter the two numbers");
scanf ("%d %d",&a,&b);
if (a>b)
goto big;
else
goto small;
big :printf ("big value is %d",a);
goto stop;
small :printf ("small value is %d",b);
goto stop;
stop;
}
```

Simple Programs Covering Above Topics

Practice Programs

1. Write a C program to find out smallest value among A, B,C.

Ans:

```
include <stdio.h>
int a,b,c;
clrscr();
scanf(%d %d %d, &a, &b, &c);
if (a<b)
        {
```

```
        if(a<c)
        printf("a is small/n")
        else
}
```

02. Write a 'C' programe for 5th multiplication table with the help of goto statement.

Ans.

```
#include<stdio.h>
main( )
{
        int t, n = 1, P;
        Printf("Enter table number:");
        Scanf("%d,&t);
        A:
        if (n<=10)
        {
                P=t * n;
                Printf("%d * %d = %d \n", t,n,p);
                n++;
                goto A;
        }
        else
                printf("Out of range");
        }
```

03. Write a 'C' program to find greatest among three numbers.

Ans.

```
#include<stdio.h>
void main( )
```

```
{
        int a,b,c;
        printf("enter the values of a,b,c,");
        scanf("%d%d%d", &a,&b,&c);
        if((a>b)&&(c>b))
        {
                if(a>c)
                printf("a is the max no");
                else
                printf("C is the max no");
        }
                else if ((b>c)&&(a>c))
        {
                if(b>a)
                printf("b is the max no");
                else
                printf("a is the max no");
        }
                else if ((b>a)&&(c>a))
        {
                if(b>c)
                printf("b is the max no");
                else
                printf("C is the max no");
        }
}
```

Model Questions

Short Answer Type Question - 2 Marks

1. Write the structure of C program
2. Define a variable and a constant in C.
3. What is an expression in C.
4. What are the operators used in C.
5. Mention the significance of main( ) function.
6. What are formatted and Unformatted Input-output statements.
7. Write the syntax of scanf() and printf() statements.
8. Write the syntax do loop control structure.
9. Write short notes on go to statement?
10. Mention difference between While loop and do…While loop.
11. What is Nested Loop?
12. Write the syntax of While statement.
13. Write the syntax of for… loop statement.
14. Write about 'Switch" Statement.
15. Write the syntax of Simple if statement.
16. Write the syntax of if … else statement.
17. What is Preprocessor statement in C.
18. What are different types of errors occurred during the execution of C program.
19. What is Variable and Constant in C? What are types of Constants in C.
20. What are basic data types in C.
21. What is String Constant.

Long Answer Type Questions - 6 Marks

01. Explain the basic structure of C program.
02. Write about data types used in C.

03. What is Constant? Explain various types of constants in C. (or)

04. Explain various types of Operators in C.

05. Explain formatted and un-formatted input and output statements in C

06. Explain various conditional control structures in C.

07. Explain various conditional looping statements in C.

08. Write the differences between Break and Continue

Note: Practice some more programs related using above statements.

UNIT 3

Functions

Structure

Learning Objectives

- Define a Function
- Stress on Return statement
- Write programs using function call techniques.
- Function prototype
- Differentiate Local and Global variables
- Recursion.

3.0 Introduction

Experienced programmer used to divide large (lengthy) programs in to parts, and then manage those parts to be solved one by one. This method of programming approach is to organize the typical work in a systematic manner. This aspect is practically achieved n C language thorough the concept known as 'Modular Programming".

The entire program is divided into a series of modules and each module is intended to perform a particular task. The detailed work to be solved by the module is described in the module (sub program) only and the main program only contains a series of modulus that are to be executed. Division of a main program in to set of modules and assigning various tasks to each module depends on the programmer's efficiency.

Whereas there is a need for us repeatedly execute one block of statements in one place of the program, loop statements can be used. But, a block of statements need to be repeatedly executed in many parts of the program, then repeated coding as well as wastage of the vital computer resource memory will wasted. . If we adopt modular programming technique, these disadvantages can be eliminated. The modules incorporated in C are called as the FUNCTIONS, and each function in the program is meant for doing specific task. C functions are easy to use and very efficient also.

3.1 Functions

Definition

A function can be defined as a subprogram which is meant for doing a specific task.

In a C program, a function definition will have name, parentheses pair contain zero or more parameters and a body. The parameters used in the parenthesis need to be declared with type and if not declared, they will be considered as of integer type.

The general form of the function is :

```
function type name <arg1,arg2,arg3, ————————,argn>)
  data type  arg1, arg2,;
  data type argn;
    {
        body of function;
```

_____

_____

_____

return (<something>);

}

From the above form the main components of function are

- Return type
- Function name
- Function body
- Return statement

Return Type

Refers to the type of value it would return to the calling portion of the program. It can have any of the basic data types such as int, float, char, etc. When a function is not supposed to return any value, it may be declared as type void

Example

void function name(- - - - - - - - - -);

int function name(- - - - - - - - - -);

char function name (— - - - - - -);

Function Name

The function name can be any name conforming to the syntax rules of the variable.

A function name is relevant to the function operation.

Example

output( );

read data( );

Formal arguments

The arguments are called **formal arguments (or) formal parameters**, because they represent the names of data items that are transferred into the function from the calling portion of the program.

Any variable declared in the body of a function is said to be local to that function, other variable which were not declared either arguments or in the function body, are considered "**globol"** to the function and must be defined externally.

Example

```
int biggest (int a, int b)
{
_____________________________
_____________________________
_____________________________
return( );
}
```

a, b are the formal arguments.

Function Body

Function body is a compound statement defines the action to be taken by the function. It should include one or more **"return"** statement in order to return a value to the calling portion of the program.

Example

```
int biggest(int a, int b)
{
if ( a > b)
return(a); body of function.
else
return(b);
}
```

Every C program consists of one or more functions. One of these functions must be called as **main.** Execution of the program will always begin by carrying out the instructions in main. Additional functions will be subordinate to main. If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition can't be embedded within another.

Generally a function will process information that is passed to it from the calling portion of the program and return a single value. Information is passed to the function via arguments (**parameters**) and returned via the **"return"** statement.

Some functions accept information but do not return anything (***ex:*** printf()) whereas other functions (***ex:*** scanf()) return multiple values.

3.1.1 The Return Statement

Every function subprogram in C will have return statement. This statement is used in function subprograms to return a value to the calling program/function. This statement can appear anywhere within a function body and we may have more than one return statement inside a function.

The general format of return statement is

```
return;
(or)
return (expression);
```

If no value is returned from function to the calling program, then there is no need of return statement to be present inside the function.

Programs using function Call Techniques

Example 1: Write a program to find factorial to the given positive integer ,using function technique.

```
# include <stdio.h>
main( )
{
    int n;
    printf ( " Enter any positive number\n");
    scanf( "%d", &n);
```

```
    printf( " The factorial of %d s %d \n",fact (n));
}
fact( i)
int I;
{
      int j; f = 1 ;
      for ( j = I; j>0; j - -)
       f  = f * I;
       return ( f ) ;
}
```

In the above program function with name 'fact' is called by the main program. The function fact is called with n as parameter. The value is returned through variable f to the main program.

Example 2: Write a program to find the value of f(x) as $f(x) = x^2 + 4$, for the given of x. Make use of function technique.

```
 # include <stdio.h>
 main( )
{
        f ( );
}
f ( )
 {  int  x,y ;
 printf( " Enter value of x \n");
 scanf( " %d", & x );
  y   = (x * x + 4);
 printf ( " The value of f (x) id %d \n", y ) ;
}
```

3.2 Differences between Function and Procedures

Procedure	Function
1. Procedure is a sub program which is included with in main program.	1. Functions is sub program which is intended for specific task. Eg. sqrt()
2. Procedure donot return a value.	2. Functions may or may not return a value.
3. Procedure cannot be called again and again.	3. Function once defined can be called any where n number of times.
4. Global variables cannot be used in procedure.	4. In functions both local and global variables can be used.
5. Procedures can be written only in procedural programming such as Dbase, Foxpro.	5. Functions can be written in modular programming such as C, C++

3.3 Advantages of Function

The main advantages of using a function are:

- Easy to write a correct small function
- Easy to read and debug a function.
- Easier to maintain or modify such a function
- Small functions tend to be self documenting and highly readable
- It can be called any number of times in any place with different parameters.

Storage class

A variable's storage class explains where the variable will be stored, its initial value and life of the variable.

Iteration

The block of statements is executed repeatedly using loops is called Iteration

Categories of Functions

A function, depending on, whether arguments are present or not and a value is returned or not.

A function may be belonging to one of the following types.

1. Function with no arguments and no return values.

2. Function with arguments and no return values.

3. Function with arguments and return values

3.4 Advanced Featured of Functions

a. Function Prototypes

b. Calling functions by value or by reference

c. Recursion.

a. Function Prototypes

The user defined functions may be classified as three ways based on the formal arguments passed and the usage of the **return** statement.

a. Functions with no arguments and no return value

b. Functions with arguments no return value

c. Functions with arguments and return value.

a. Functions with no arguments and no return value

A function is invoked without passing any formal arguments from the calling portion of a program and also the function does not return back any value to the called function. There is no communication between the calling portion of a program and a called function block.

Example:

```
#include <stdio.h>
main()
{
        void message( ); Function declaration
        message( ); Function calling
}
```

```
void message( )
{
        printf (“GOVT JUNIOR  COLLEGE \n”);
        printf (“\t HYDERABAD”);
}
```

b. Function with arguments and no return value

This type of functions passes some formal arguments to a function but the function does not return back any value to the caller. It is any one way data communication between a calling portion of the program and the function block.

Example

```
#include <stdio.h>
main()
{
        void square(int);
        printf (“Enter a value for n \n”);
        scanf (“%d”,&n);
        square(n);
}
void square (int n)
{
        int value;
        value = n * n;
        printf (“square of %d is %d “,n,value);
}
```

c. Function with arguments and return value

The third type of function passes some formal arguments to a function from a calling portion of the program and the computer value is transferred back to the caller. Data are communicated between the calling portion and the function block.

Example

```
#include <stdio.h>
main()
{
        int square (int);
        int value;
        printf (“enter a value for n \n”);
        scanf(“%d”, &n);
        value = square(n);
        printf (“square of %d is %d “,n, value);
}
int square(int n)
{
        int p;
        p = n * n;
        return(p);
}
```

The keyword **VOID** can be used as a type specifier when defining a function that does not return anything or when the function definition does not include any arguments.

The presence of this keyword is not mandatory but it is good programming practice to make use of this feature.

Actual and Formal Parameters (or) Arguments

Function parameters are the means of communication between the calling and the called functions. The parameters may classify under two groups.

1. Formal Parameters
2. Actual Parameters

1. Formal Parameters

The formal parameters are the parameters given in function declaration and function definition. When the function is invoked, the formal parameters are replaced by the actual parameters.

2. Actual Parameters

The parameters appearing in the function call are referred to as actual parameters. The actual arguments may be expressed as constants, single variables or more complex expression. Each actual parameter must be of the same data type as its corresponding formal parameters.

Example

```
#include <stdio.h>
int sum (int a , int b )
{
        int c;
        c = a + b;
        return(c);
}
main( )
{
        intx,y,z;
        printf (“enter value for x,y \n”);
        scanf (“%d %d”,&x,&y);
        z = x + y;
        printf (“ sum is = %d”,z);
}
```

The variables **a and b** defined in function definition are known as **formal parameters**. The variables **x and y** are **actual parameters**.

Local and Global Variable:

The variables may be classified as local or global variables.

Local Variable

The variables defined can be accessed only within the block in which they are declared. These variables are called "Local" variables

Example

```
funct (int ,int j)
{
        intk,m;
        ————————;
        ————————;
}
```

The integer variables **k** and **m** are defined within a function block of the "**funct()"**. All the variables to be used within a function block must be either defined at the beginning of the block or before using in the statement. Local variables one referred only the particular part of a block of a function.

Global Variable

Global variables defined outside the main function block. Global variables are not contained to a single function. Global variables that are recognized in two or more functions. Their scope extends from the point of definition through the remainder of the program.

b. Calling functions by value or by reference

The arguments are sent to the functions and their values are copied in the corresponding function. This is a sort of information inter change between the calling function and called function. This is known as Parameter passing. It is a mechanism through which arguments are passed to the called function for the required processing. There are two methods of parameter passing.

1. Call by Value
2. Call by reference.

1. Call by value: When the values of arguments are passed from calling function to a called function, these values are copied in to the called function. If

any changes are made to these values in the called function, there are NOCHANGE the original values within the calling function.

Example

```
#include <stdio.h>
main();
{
        int n1,n2,x;
        int cal_by_val();
        N1 = 6;
        N2 = 9;
        printf( n1 = %d and  n2= %d\n", n1,n2);
        X = cal_by_Val(n1,n2);
        Printf( n1 = %d and  n2= %d\n", n1,n2);
        Printf(" x= %d\n", x);
                / * end of main*/
        /*function to illustrate call by value*/
        Cal_by_val(p1,p2)
        int p1,p2;
        {
                int  sum;
                Sum = (p1 + p2);
                P1 + = 2;
                P2* = p1;
                printf( p1 = %d and  p2= %d\n", p1,p2);
                return( sum);
        }
}
```

When the program is executed the output will be displayed

N1 = 6 and n2 = 9

P1 = 8 and p2 = 72

N1 = 6 and n2 = 9

X = 15

There is NO CHANGE in the values of n1 and n2 before and after the function is executed.

2. Cal by Reference: In this method, the actual values are not passed, instead their addresses are passed. There is no copying of values since their memory locations are referenced. If any modification is made to the values in the called function, then the original values get changed with in the calling function. Passing of addresses requires the knowledge of pointers.

Example

This program accepts a one-dimensional array of integers and sorts them in ascending order. [This program involves passing the array to the function].

```
# include <stdio.h>
main();
{
int num[20], I,max;
void sort_nums();
printf( " enter the size of the array"\n");
scanf("%d", &max);
for( i=0; i<max;I++)
sort_nums(num,max)        /* Function reference*/
printf("sorted numbers are as follows\n");
for( i=0; i<max;I++)
printf("%3d\n",num[i]);
/* end of the main*/
/* function to sort list of numbers*/
```

```
Void sort_nums(a,n)
Int a[],n;
{
        Int I,j,dummy;
        For(i=0;i<n;i++)
        {
                For(j=0;  j<n;  j++)
                {
                        If ( a[i] >a[j])
                        {
                                Dummy = a[i];
                                a[i] = a[j];
                                a[j] = dummy;
                        }
                }
        }
}
```

3.5 Recursion

One of the special features of C language is its support to recursion. Very few computer languages will support this feature.

Recursion can be defines as the process of a function by which it can call itself. The function which calls itself again and again either directly or indirectly is known as recursive function.

The normal function is usually called by the main ( ) function, by mans of its name. But, the recursive function will be called by itself depending on the condition satisfaction.

For Example,

```
main ( )
{
```

```
        f1( ) ;                    ——— Function called by main
        _____________
        _____________
          _____________
}
f1( ) ;                    ——— Function definition
{
        _____________
        _____________
        _____________
          f1( ) ;                  ——— Function called by itself
}
```

In the above, the main ( ) function s calling a function named f1( ) by invoking it with its name. But, inside the function definition f1( ), there is another invoking of function and it is the function f1( ) again.

Example programs on Recursion

Example 1 : Write a program to find the factorial of given non-negative integer using recursive function.

```
#include<stdio.h>
main ( )
{
        int result, n;
        printf( " Enter any non-negative integer\n");
        scanf ( " %d", & n);
        result  =  fact(n);
        printf (  " The factorial of %d is %d \n", n, result);
}
```

```
fact( n )
int  n;
{
        int  i ;
        i = 1;
        if ( i = = 1) return ( i);
        else
        {
                i   =  i * fact ( n - 1);
                return ( i );
        }
}
```

Example 2: Write 'C' program to generate Fibonacci series up to a limit using recursion function. .

```
#include<stdio.h>
#include<conio.h>
int Fibonacci (int);
void main ( )
{
        int i, n;
        clrscr ( );
        printf ("Enter no. of Elements to be generated" \n)
        scanf ("%d", &n);
        for (i=1; i<n; i++)
        printf ("%d", Fibonacci (i));
        getch( );
}
```

```
int Fibonacci (int n)
{
        int fno;
        if (n= =1)
        return 1;
        else
                if (n= =2);
                  return 1;
        else
                fno=Fibonacci (n-1) + Fibonacci (n-2);
        return fno;
}
```

Model Questions

Short Answer Type Question - 2 Marks

01. What is function? Write its syntax.
02. What is I/O function? List different types of I/O functions
03. What are the advantages of functions?
04. Write differences between Global and Local variables.
05. List categories of functions
06. What is storage class?
07. What is Iteration
08. What is recursion**?**

Long Answer Type Question - 6 Marks

01. What is Function? Explain in detail
02. Explain different types of functions with an example.
03. Explain about I/O functions.
04. Discuss about Global and Local variables.

05. Explain about Call-by-value with an example.
06. Explain about Call-by-reference with an example.
07. Discuss about Functions and Procedures..

UNIT 4

Arrays in 'C'

Structure

Learning Objectives

- To understand the importance of Array
- Definition of array
- Declaration and Initializing of an Array
- Types of Arrays
- Examples of an Array

4.0 Introduction

If we deal with similar type of variables in more number at a time, we may have to write lengthy programs along with long list of variables. There should be

more number of assignment statements in order to manipulate on variables. When the number of variables increases, the length of program also increase.

In the above situations described above, where more number of same types of variables is used, the concept of ARRAYS is employed in C language. These are very much helpful to store as well as retrieve the data of similar type.

An Array describes a contiguously allocated non-empty set of objects with the same data type. The using arrays many number of same type of variables can be grouped together. All the elements stored in the array will referred by a common name.

4.1 Definition of Array

Array can be defined as a collection of data objects which are stored in consecutive memory locations with a common variable name.

OR

Array can be defined as a group of values referred by the same variable name.

OR

An Array can be defined as a collection of data objects which are stored in consecutive memory locations with a common variable name.

The individual values present in the array are called elements of array. The array elements can be values or variables also.

4.2 Types of Arrays

Basically arrays can divide in to

1. One Dimensional Array

An array with only one subscript is called as *one-dimensional array or 1-d array*. It is used to store a list of values, all of which share a common name and are separable by subscript values

2. Two Dimensional Array

An array with two subscripts is termed as two-dimensional array.

A two-dimensional array, it has a list of given variable -name using two subscripts. We know that a one-dimensional array can store a row of elements, so, a two-dimensional array enables us to store multiple rows of elements.

4.2.1 Initialization of Array

Array can be made initialized at the time of declaration itself. The general form of array initialization is as below

type name[n] = [element1, element2, …. element n];

The elements 1, element2… element n are the values of the elements in the array referenced by the same.

Example1:- int codes[5] = [12,13,14,15,16];

Example2:- float a[3] = [1.2, 1.3, 1.4];

Example3:- char name [5] = ['S', 'U', 'N', 'I', 'L'];

In above examples, let us consider one, it a character array with 5 elements and all the five elements area initialized to 5 different consecutive memory locations.

name[0] = 'S'

name[1] = 'U'

name[2] = 'N'

name[3] = 'I'

name[4] = 'L'

Rules for Array Initialization

1. Arrays are initialized with constants only.
2. Arrays can be initialized without specifying the number of elements in square brackets and this number automatically obtained by the compiler.
3. The middle elements of an array cannot be initialized. If we want to initialize any middle element then the initialization of previous elements is compulsory.
4. If the array elements are not assigned explicitly, initial values will be set to zero automatically.
5. If all the elements in the array are to be initialized with one and same value, then repletion of data is needed.

4.2.2 Declaration of Array

The array must be declared as other variables, before its usage in a C program. The array declaration included providing of the following information to C compiler.

- The type of the array (ex. int, float or char type)
- The name of the array (ex A[ ],B[ ] , etc)
- Number of subscripts in the array (i.e whether one – dimensional or Two-dimensional)
- Total number of memory locations to be allocated.

The name of the array can be kept by the user (the rule similar to naming to variable).

There is no limit one on dimensions as well as number of memory locations and it depends o the capacity of computer main memory..

The general form for array declaration is

Type name[n] ; { one dimensional array}

Ex : int marks[20];

char name[15];

float values [10];

An array with only one subscript is called as *one-dimensional array or 1-d array*. It is used to store a list of values, all of which share a common name and are separable by subscript values.

Declaration of One-dimensional Arrays:

The general form of declaring a one-dimensional array is

data-type	array-name [size];

Where data-type refers to any data type supported by C, array-name should be a valid C identifier; the size indicates the maximum number of storage locations (elements) that can be stored.

Each element in the array is referenced by the array name followed by a pair of square brackets enclosing a subscript value. The subscript value is indexed

from 0 to size -1. When the subscript vale is 0, first element in the array is selected, when the subscript value is 1, second element is selected and so on.

Example

int x [6];

Here, x is declared to be an array of int type and of size six. Six contiguous memory locations get allocated as shown below to store six integer values.

1	2	3	4	5	6
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Each data item in the array x is identified by the array name x followed by a pair of square brackets enclosing a subscript value. The subscript value is indexed from 0 to 5. i.e., x[0] denotes first data item, x[1] denotes second data item and x[5] denotes the last data item.

Initialization of One-Dimensional Arrays:

Just as we initialize ordinary variables, we can initialize one-dimensional arrays also, i.e., locations of the arrays can be given values while they are declared.

The general form of initializing an array of one-dimension is as follows:

data - type array - name [size] = {list of values};

The values in the list are separated by commas.

Example

int x [6] = {1, 2, 3, 4, 5, 6 };

as a result of this, memory locations of x get filled up as follows:

1	2	3	4	5	6
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Points to be considered during the declaration

1. If the number of values in initialization value - is less then the size of an array, only those many first locations of the array are assigned the values. The remaining locations are assigned zero.

Example: int x [6] = {7, 8, 6 };

The size of the array x is six, initialization value - consists of only three locations get 0 assigned to them automatically, as follows:

7	8	6	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

2. If the number of values listed within initialization value - list for any array is greater than the six the array, compiler raises an error.

Example:

int x [6] = {1, 2, 3, 4, 5, 6, 7, 8 };

The size of the array x is six. But the number of values listed within the initialization - list is eight. This is illegal.

3. If a static array is declared without initialization value - list then the all locations are set to zero.

Example:

static int x [6];

0	0	0	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

4. If size is omitted in a 1-d array declaration, which is initialized, the compiler will supply this value by examining the number of values in the initialization value - list.

Example:

int x [ ] = {1, 2, 3, 4, 5, 6 };

0	0	0	0	0	0
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Since the number of values in the initialization value-list for the array x is six, the size of x is automatically supplied as six.

5. There is no array bound checking mechanism built into C-compiler. It is the responsibility of the programmer to see to it that the subscript value does not go beyond size-1. If it does, the system may crash.

Example:

int x [6];

x [7] = 20;

Here, x [7] does not belong to the array x, it may belong to some other program (for example, operating system) writing into the location may lead to unpredictable results or even to system crash.

6. Array elements can not be initialized selectively.

Example:

An attempt to initialize only 2nd location is illegal, i.e.,

int x [6] = { , 10 } is illegal.

Similar to arrays of int type, we can even declare arrays of other data types supported by C, also.

Example

char ch [6];

ch[0]	ch[1]	ch[2]	ch[3]	ch[4]	ch[5]

ch is declared to be an array of char type and size 6 and it can accommodate 6 characters.

Example:

float x [6];

x is declared to be an array of float type and size 6 and it can accommodate 6 values of float type. Following is the scheme of memory allocation for the array x:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]

Note

A one array is used to store a group of values. A loop (using, for loop) is used to access each value in the group.

Example

Program to illustrate declaration, initialization of a 1-d array

#include<stdio.h>

```
#include<conio.h>
void main( )
{
        int i, x [6] = {1, 2, 3, 4, 5, 6 };
        clrscr( );
        printf("The elements of array x \n");
        for(i=0; i<6; i++)
                printf("%d", x [i]);
        getch( );
}
```

Input – Output:

The elements of array x

1 2 3 4 5 6

4.3 Two - Dimenstional Array

An array with two subscripts is termed as two-dimensional array.

A two-dimensional array, it has a list of given variable -name using two subscripts. We know that a one-dimensional array can store a row of elements, so, a two-dimensional array enables us to store multiple rows of elements.

Example*:* Table of elements or a Matrix.

Syntax of two-dimensional arrays:

The syntax of declaring a two-dimensional array is:

```
data - type array - name [rowsize] [ colsize];
```

Where, data-type refers to any valid C data type, array -name refers to any valid C identifier, row size indicates the number of rows and column size indicates the number of elements in each column.

Row size and column size should be integer constants.

Total number of location allocated = (row size * column size).

Each element in a 2-d array is identified by the array name followed by a pair of square brackets enclosing its row-number, followed by a pair of square brackets enclosing its column-number.

Row-number range from 0 to row size-1 and column-number range from 0 to column size-1.

Example: int y [3] [3];

y is declared to be an array of two dimensions and of data type Integer (int), row size and column size of y are 3 and 3, respectively. Memory gets allocated to store the array y as follows. It is important to note that y is the common name shared by all the elements of the array.

Column numbers

Row numbers	0	1	2
1	x[0] [0]	x[0] [1]	x[0] [2]
2	x[1] [0]	x[1] [1]	x[1] [2]
3	x[2] [0]	x[2] [1]	x[2] [2]

Each data item in the array y is identifiable by specifying the array name y followed by a pair of square brackets enclosing row number, followed by a pair of square brackets enclosing column number.

Row-number ranges from 0 to 2. that is, first row is identified by row-number 0, second row is identified by row-number 1 and so on.

Similarly, column-number ranges from 0 to 2. First column is identified by column-number 0, second column is identified by column-number 1 and so on.

x [0] [0] refers to data item in the first row and first column

x [0] [2] refers to data item in the first row and third column

x [2] [3] refers to data item in the third row and fourth column

x [3] [4] refers to data item in the fourth row and fifth column

Initialization of Two-Dimensional Array

There are two forms of initializing a 2-d array.

First form of initializing a 2-d array is as follows:

```
data - type array - name [rowsize][colsize] = [initializer - list];
```

Where, data-name refers to any data type supported by C. Array-name refers to any valid C identifier. Row size indicates the number of rows, column size indicates the number of columns of the array. initialier-list is a comma separated list of values.

If the number of values in initializer-list is equal to the product of row size and column size, the first row size values in the initializer-list will be assigned to the first row, the second row size values will be assigned to the second row of the array and so on

Example: int x [2] [4] = {1, 2, 3, 4, 5, 6, 7, 8 };

Since column size is 4, the first 4 values of the initializer-list are assigned to the first row of x and the next 4 values are assigned to the second row of x as shown hereinafter

1 2 3 4

5 6 7 8

Note: If the number of values in the initializer-list is less than the product of rowsize and colsize, only the first few matching locations of the array would get values from the initializer-list row-wise. The trailing unmatched locations would get zeros.

Example: int x [2] [4] = {1, 2, 3, 4};

The first row of x gets filled with the values in the initializer-list. The second row gets filled with zeros.

1	2	3	4
0	0	0	0

The second form of initializing a 2-d array is as follows:

data-type array-name [rowsize] [colsize]

= { {initializer-list1},

{initializer-list2}, ……};The values in initializer-list 1 are assigned to the locations in the first row. The values in initializer-list2 are assigned to the locations in the second row and so on.

Example: int x [2] [4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };

As a result of this, the array x gets filled up as follows:

1	2	3	4
5	6	7	8

Note

1. If the number of values specified in any initializer-list is less than colsize of x, only those may first locations in the corresponding row would get these values. The remaining locations in that row would get 0.

Example: int x [2] [4] = { { 1, 2, 3 }, { 4, 5, 6, 7 } };

Since the first initializer-list has only three values, x [0] [0] is set to 1, x [0] [1] is set to 2, x [0] [2] is set to 3 and the fourth location in the first row is automatically set to 0.

1	2	3	0
4	5	6	7

2. If the number of values specified in any initializer-list is more than colsize of x, compiler reports an error.

Example: int x [2] [4] = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9 } };

colsize is 4, but the number of values listed in the first row is 5. This results in compilation error.

3. Array elements can not be initialized selectively.

4. It is the responsibility of the programmer to ensure that the array bounds do not exceed the rowsize and colsize of the array. If they exceed, unpredictable results may be produced and even the program can result in system crash sometimes.

5. A 2-d array is used to store a table of values (matrix).

Similar to 2-d arrays of int type, we can declare 2-arrays of any other data type supported by C, also.

Example: float x [4] [5];

x is declared to be 2-d array of float type with 4 rows and 5 columns double y [4] [5];

y is declared to be 2-d arrays of double type with 4 rows and 5 columns

Note

A two-dimensional array is used to store a table of values. Two loops (using for loops) are used to access each value in the table, first loop acts as a row selector and second loop acts as a column selector in the table.

4.4 Declare, Initialize Array of Char Type

Declaration of Array of char type: A string variable is any valid C variable name and is always declared as an array. The syntax of declaration of a string variable is:

char string-name[size];

The size determines the number of character in the string name.

Example: An array of char type to store the above string is to be declared as follows:

char str[8];

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]	str[7]
P	r	o	g	r	a	m	\0

An array of char is also called as a string variable, since it can store a string and it permits us to change its contents. In contrast, a sequence of characters enclosed within a pair of double quotes is called a string constant.

Example: "Program" is a string constant.

Initialization of Arrays of char Type

The syntax of initializing a string variable has two variations:

Variation 1

char str1 [6] = { 'H', 'e', 'l', 'l', 'o', '\0'};

Here, str1 is declared to be a string variable with size six. It can store maximum six characters. The initializer – list consists of comma separated character constants. Note that the null character '\0' is clearly listed. This is required in this variation.

Variation 2

char str2 [6] = { "Hello" };

Here, str2 is also declared to be a string variable of size six. It can store maximum six characters including null character. The initializer-list consists of a

string constant. In this variation, null character '\0' will be automatically added to the end of string by the compiler.

In either of these variations, the size of the character array can be skipped, in which case, the size and the number of characters in the initializer-list would be automatically supplied by the compiler.

Example

char str1 [ ] = { "Hello" };

The size of str1 would be six, five characters plus one for the null character '\0'.

char str2 [ ] = { 'H', 'e', 'l', 'l', 'o', '\0'};

The size of str2 would be six, five characters plus one for null character '\0'.

Example 1

Program to sort a list of numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int x [6], n, i, j, tmp;
        clrscr( );
        printf ("Enter the no. of elements \n");
        scanf ("%d", & n);
        printf ("Enter %d numbers \n", n);
        for (i=0; i<n; i++)
        scanf("%d", & x [i]);
        /* sorting begins */
        for (i=0; i<n; i++)
        for (j=i + 1; j<n; j++)
        if (x[i]>x[j])
```

```
        {
                tmp = x [i];
                x [i] = x [j];
                x [j] = tmp;
        }
        /* sorting ends */
        printf ("sorted list \n");
        for (i=0; i<n; i++)
        printf ("%d", x [i]);
        getch( );
}
```

Input – Output

```
Enter the no. of elements
5
Enter 5 numbers
10      30      20      50      40
Sorted list
10 20      30      40      50
```

Example 2 :

C program for addition of two matrices.

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main( )
{
        int x [5] [5], y [5] [5], z [5] [5], m, n, p, q, i, j;
```

```
clrscr ( );
printf (“Enter number of rows and columns of matrix x \n”);
scanf (“%d %d”, &m, &n);
printf (“Enter number of rows and columns of matrix y \n”);
scanf (“%d %d”, &p, &q);
if ((m ! =P) | | (n!=q))
{
        printf (“Matrices not compatible for addition \n”);
        exit(1);
}
printf (“Enter the elements of x \n”);
for (i=0; i<m; i++)
for (j=0; j<n; j++)
scanf (“%d”, &x [i][j]);
printf (“Enter the elements of x \n”);
for (i=0; i<p; i++)
for (j=0; j<n; j++)
scanf (“%d”, &y [i] [j]);
/* Summation begins */
for (i=0; j<m; i++)
for (j=0; j<n; j++)
z[i] [j] = x [i] [j] + y [i] [j];
/* summation ends */
printf (“Sum matrix z \n);
for (i=0; i<m; i++)
{
        for (j=0; j<n; j++)
```

```
            printf ("%d", z[i] [j]);
            printf ("\n");
        }
        getch( );
    }
```

Example3

Write a program for multiplication of two matrices.

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main( )
{
    int x [5] [5], y [5] [5], z [5] [5], m, n, p, q, I, j, k;
    clrscr ( );
    printf ("Enter number of rows and columns of matrix x \n");
    scanf ("%d %d", &m, &n);
    printf ("Enter number of rows and columns of matrix y \n");
    scanf (%d %d", &p, &q);
    if (n!=p)
    {
        printf ("Matrices not compatible for multiplication \n");
        exit (1);
    }
    printf ("Enter the elements of x \n");
    for (i=0; i<m; i++)
    for (j=0; j<n; j++)
    scanf ("%d", &x [i] [j]);
```

```
        printf ("Enter the elements of y \n");
        for (i=0; i<p; i++)
        for (j=0; j<q; j++)
        scanf ("%d", &x [i] [j]);
        printf ("Enter the elements of y \n");
        for (i=0; i<p; i++)
        for (j=0; j<q; j++)
        scanf ("%d", &y [i] [j]);
        /* Multiplication of matrices of x 7 y ends */
        for (i=0; i<m; i++)
        for (j=0; j<q; j++)
        { z [i] [j] = 0;
        for (k=0; k<n; k++)
        z [i] [j] + = x [i] [k] * y [k] [j];
        }
        /* Multiplication of matrices of x & y ends */
        printf ("Product Matrix z \n");
        for (i=0; i<m; i++)
        {
                for (j=0; j<q; j++)
                        printf ("%d", z[i] [j]);
                printf ("\n");
        }
        getch( );
}
```

Example 4: C program to print transpose of a matrix.

[The transpose of a matrix is obtained by switching the rows and columns of matrix].

```
#include<stdio.h>
#include<conio.h>
void main ( )
{
        int a[3] [3], b[3] [3], i, j;
        clrscr ( );
        printf ("Enter the elements of the matrix : \n");
        for (i=0; i<3; i++)
        for (j=0; j<3; j++)
        scanf ("%d", &a[i] [j]);
        printf ("given matrix is : \n");
        for (i=0; i<3; i++)
        {
                for (j=0; j<3; j++)
                        printf ("%d", a[i] [j]);
                        printf ("\n");
        }
        printf ("transpose of given matrix is : \n");
        for (i=0; i<3; i++)
        {
                for (j=0; j<3; j++)
                {
                        b [i] [j] = a [j] [i];
                        printf ("%d", b[i] [j]);
```

```
                printf ("\n");
            }
        }
        getch( );
    }
```

Example 5

Write a 'C' program to find the average marks of 'n' students of five subjects for each subject using arrays.

Ans.

```
#include <stdio.h>
void main( )
{
    int Sno, S1,S2,S3;
    float tot, avg;
    char sname[10];
    printf("Enter student no;");
    scanf("%d", & Sname);
    printf("Enter subject - 1, sub - 2, sub - 3 marks;");
    scanf("%d %d %d", &s1,&s2,&s3);
    tot = S1+S2+S3;
    avg = tot/3;
        printf("total = %f", tot);
        printf("Average = %f", avg);
}
```

Example 6

Write a C program to check the given word is 'Palindrome' or not.

Ans.

```
#include<stdio.h>
```

```
#include<conio.h>
void main ( )
{
        char str[80], rev[80];
        int k, i, j, flag = 0;
        clrscr ( );
        printf (“Enter any string (max. 80 chars) : \n”);
        gets (str);
for (i=0; str[i]!= ‘\0’; i++);
        for (j=i-1; k=0; j>=0; j—, k++)
                rev[k] = str[j];
        rev[k] = ‘\0’;
        for (i=0; str[i]!= ‘\0’; i++)
        {
                if (str[i]!=rev[i])
                {
                        flag=1;
                        break;
                }
        }
                if (flag = =1);
                        printf (“Given string is not palindrome. \n”);
        else
                        printf (“Given string is palindrome. \n”);
        getch( );
}
```

4.5 String Handling Functions in 'C'

To perform manipulations on string data, there are built-in-fraction (library function) Supported by 'c' compiler. The string functions are.

1. STRLEN (S1)
2. STRCMP (S1, S2)
3. STRCPY (S1, S2)
4. STRCAT (S1, S2)

1. STRLEN (S1): This function is used to return the length of the string name S1,

Ex: S1 = 'MOID'

STRLEN (S1) = 4

2. STRCMP (S1, S2): This is a library function used to perform comparison between two strings. This function returns a value <zero when string S1 is less than S2. The function return a value 0 whenS1=S2. Finally the function return a value > 0 when S1>S2.

3. STRCPY (S1, S2): This is a library function used to copy the string S2 to S1.

4. STRCAT (S1, S2): This is a library function used to join two strings one after the other. This function concatenates string S2 at the end of string S1.

Example5 : C program to concatenate the given two strings and print new string.

```
#include<stdio.h>
#include<conio.h>
main ( )
{
        char s1[10], s2[10], s3[10],
        int i,j,k;
        printf ("Enter the first string : \n");
        scanf("%s,S",s1);
        printf ("Enter the second string : \n");
```

```
        scanf("%s,S",s2);
        i = strlen(s1);
        j = strlen(s2);
        for (k=0,k<i,k++)
        s3[k] = s1[k];
        for (k=0,k<j,k++)
        s3[i+k] = s2[k];
        s3[i+j] = \0';
        printf(" The new string is \n".s3);
}
```

4.6 File Operations like fopen(), fclose(), fprint(), fscan()

The help of these functions the user can open a file with the data file specifications, create and write information in to the file and can close the file. The following are the file processing related functions.

a. FILE OPEN function fopen()

b. FILE CLOSE function fclose()

c. FILE INPUT functions getc() and fscanf()

d. FILE OUTPUT functions putc() and fprintf()

a. The function fopen()

This function is used to open a data file. Moreover, this function returns a pointer to a file. The use of the function is

file pointer = fopen(filename, mode);

Where file pointer is a pointer to a type FILE, the file name of the file name is name of the file in which data is stored or retrieved (should be enclosed within double quotes) and the mode denotes the type of operations to be performed on the file (this also to be enclosed within double quotes).

But, before making this assignment ,the file pointer and fopen() should be declared as FILE pointer type variables as under :

FILE *file pointer, * fopen() ;

The mode can be one of the following types.

MODE		MEANING
"r "	-	read from the file
"w"	-	write to the file
"a"	-	append a file ie new data is added to the end of file
" r+"	-,	open an existing file for the sake of updation.
"w +"	-	create a new file for reading and writing
"a +"	-	open a file for append, create a new one if the file does not exist already

Examples

1. fptr = fopen("rk. Dat","w");

2. file= fopen("sample.dat", "r +");

b. The functions fclose ()

The files that are opened should be closed after all the desired operations are performed on it .This can be achieved through this function .The usage of this function is:

fclose (file pointer);

Where file pointer is returned value of the fopen () function.

Examples:

1. fclose (input file); 2. fclose (output file);

c. The functions getc() & fscanf()

1. getc ( ) functions: this function is used a single character from a given file ,whenever a file is referenced by a file pointer. The usage of this function is

getc (file pointer);

2. fscanf ( )function : This function is used to read a formatted data from a specified file. The general usage of this function is

fscanf (f ptr, "Control String", & list); where

fptr ⟶ a file pointer to receive formatted data

Control string ⟶ data specifications list

List ⟶ the list of variables to be read

fscaf (infile , "%d %d ," & no, &marks);

d. The functions putc()& fprint()

Example

1. putc( ) function: This function is used write a single character into a file referenced by the file pointer. The usage of this function is

putc (ch, file pointer),

Where

ch - the character to be written

file pointer - a file pointer to the file that receives the character.

2. fprintf () function: this function is used t write a for matted data in to a given file. The specified information is written on the specified file.

The general form of usage for this function is:

fprintf (fptr, "Control String", list):

Where

Fptr ⟶ file pointer to write a formatted data

Control string ⟶ data specifications list

list- ⟶ list of variables to be written.

Example

fprintf (out file,"%d %f", basic , gross);

Model Questions

Short Answer Type Questions - 2 Marks

01. What is array?

02. What are the different types of arrays?

03. How to declare an array

04. Write a C program to print "IPE" using one dimensional array

05. What is two-dimensional array? Write its application.

06. What is the String handling functions.

Long Answer Type Questions - 6 Marks

01. Define an Array. Explain one-dimensional array
02. Explain about two - dimensional array?
03. Explain how to declare, initialize array of char type.
04. Write a C program to sort a list of numbers.
05. Write a C program for addition of two matrices.
06. Write a program for multiplication of two matrices.
07. Write a program to print transpose of a matrix.
08. Explain String handling functions in C.
09. Write a C program to concatenate the given two strings and print new string.
10. Explain about File operations like fopen( ), fclose( ), fprint( ), fscan( ).

UNIT 5

Structures in 'C'

Structure

Learning Objectives

- Importance of structures
- Definition of structure
- Implementation of arrays in structures
- Definition of Union
- Union declaration
- Differences between structures and Union

5.0 Introduction

Arrays are useful to refer separate variables which are the same type. i.e. Homogeneous referred by a single name. But, we may have situations where there is a need for us to refer to different types of data (Heterogeneous) in order to derive meaningful information.

Let us consider the details of an employee of an organization. His details include employer's number (Integer type), employee's name (Character type) , basic pay (Integer type) and total salary (Float data type) . All these details seem to be of different data types and if we group them together, they will result in giving useful information about employee of the organization.

In above said situations, C provides a special data types called Structure, which is highly helpful to organize different types of variables under a single name.

5.1 Definition of Structure

A group of one or more variables of different data types organized together under a single name is called **Structure.**

Or

A collection of heterogeneous (dissimilar) types of data grouped together under a single name is called a **Structure**.

A structure can be defined to be a group of logically related data items, which may be of different types, stored in contiguous memory locations, sharing a common name, but distinguished by its members.

Hence a structure can be viewed as a heterogeneous user-defined data type. It can be used to create variables, which can be manipulated in the same way as variables of built-in data types. It helps better organization and management of data in a program.

When a structure is defines the entire group s referenced through the structure name. The individual components present in the structure are called structure members and those can be accessed and processed separately.

5.2 Structure Declaration

The declaration of a structure specifies the grouping of various data items into a single unit without assigning any resources to them. The syntax for declaring a structure in C is as follows:

struct Structure Name

```
{
        Data Type    member-1;
        Data Type    member-2;
          ….      ….
        DataType     member-n;
};
```

The structure declaration starts with the structure header, which consists of the keyword '**struct'** followed by a tag. The tag serves as a structure name, which can be used for creating structure variables. The individual members of the structure are enclosed between the curly braces and they can be of the similar or dissimilar data types. The data type of each variable is specified in the individual member declarations.

Example:

Let us consider an employee database consisting of employee number, name, and salary. A structure declaration to hold this information is shown below:

```
struct employee
{
        int eno;
        char name [80];
        float sal;
};
```

The data items enclosed between curly braces in the above structure declaration are called structure elements or structure members.

Employee is the name of the structure and is called structure tag. Note that, some members of employee structure are integer type and some are character array type.

The individual members of a structure can be variables of built – in data types (int, char, float etc.), pointers, arrays, or even other structures. All member names within a particular structure must be different. However, member names may be the same as those of variables declared outside the structure. The individual members cannot be initialized inside the structure declaration.

Note

Normally, structure declarations appear at the beginning of the program file, before any variables or functions are declared.

They may also appear before the main ( ), along with macro definitions, such as #define.

In such cases, the declaration is global and can be used by other functions as well.

5.2.1 Structure Variables

Similar to other types of variables, the structure data type variables can be declared using structure definition.

```
struct
{
    int    rollno;
    char  name[20];
    float   average;
    a, b;
}
```

In the above structure definition, a and b are said to be structure type variables. 'a' is a structure type variable containing rollno, name average as members, which are of different data types. Similarly 'b' is also a structure type variable with the same members of 'a '.

5.2.2 Structure Initialization

The members of the structure can be initialized like other variables. This can be done at the time of declaration.

Example 1

```
struct
{
    int     day;
    int     month;
    int     year;
```

```
    }
date = { 25,06,2012};
```

i.e

```
date. day = 25
date. month = 06
date. year = 2012
```

Example 2

```
struct address
{
    char    name [20];
    char    desgn [10];
    char    place [10];
} ;
```

i.e

```
struct address my-add = { 'Sree', 'AKM', 'RREDDY');
```

i.e

```
my-add . name = 'Sree'
my-add . desgn = AKM
my-add . place = RREDDY
```

As seen above, the initial values for structure members must be enclosed with in a pair of curly braces. The values to be assigned to members must be placed in the same order as they are specified in structure definition, separated by commas. If some of the members of the structure are not initialized, then the c compiler automatically assigns a value 'zero' to them.

Accessing of Structure Members

As seen earlier, the structure can be individually identified using the period operator (.). After identification, we can access them by means of assigning some values to them as well as obtaining the stored values in structure members. The following program illustrates the accessing of the structure members.

Example: Write a C program, using structure definition to accept the time and display it.

```
/ * Program to accept time and display it */
# include <stdio.h>
main( )
{
    struct
{
    int hour, min;
    float seconds;
  } time;
    printf ( "Enter  time in Hours, min and Seconds\n");
    scanf ( " %d %d %f", &time . hour, & time . min, & time . seconds);
    printf ( " The accepted time is  %d %d %f " , time . hour, time . min, time
. seconds ");
}
```

5.2.3 Nested Structures

The structure is going to certain number of elements /members of different data types. If the members of a structure are of structure data type, it can be termed as structure with structure or nested structure.

Examaple

```
struct
{
    int    rollno;
    char  name[20];
    float   avgmarks;
    struct
    {
```

```
    int  day, mon, year;
    } dob'
   } student;
```

In the above declaration, student is a variable of structure type consisting of the members namely rollno, name, avgmarks and the structure variable dob.

The dob structure is within another structure **student** and thus structure is nested. In this type of definitions, the elements of the require structure can be referenced by specifying appropriate qualifications to it, using the period operator (.) .

For example, **student. dob. day** refers to the element day of the inner structure dob.

5.3 Structures and Arrays

Array is group of identical stored in consecutive memory locations with a single / common variable name. This concept can be used in connection with the structure in the following ways**.**

a. Array of structures

b. structures containing arrays (or) arrays within a structure

c. Arrays of structures contain arrays.

5.3.1 Array of Structures

Student details in a class can be stored using structure data type and the student details of entire class can be seen as an array of structure.

Examaple

```
struct student
{
   int  rollno;
   int   year;
  int tmarks;
}
struct student class[40];.
```

In the above class [40] is structure variable accommodating a structure type student up to 40.

The above type of array of structure can be initialized as under

```
struct  student class [2] = { 001,2011,786},
                            { 002, 2012, 710}
                              };
```

i.e class[0] . rollno = 001

class[0] . year = 2011

class[0] . tmarks = 777 and

class[1] . rollno = 002

class[1] . year = 2012

class[1] . tmarks = 777 .

5.3.2 Structures containing Arrays

A structure data type can hold an array type variable as its member or members. We can declare the member of a structure as array data type similar to int, float or char.

Example

```
struct employee
{
        char ename [20];
        int eno;
};
```

In above, the structure variable employee contains character array type ename as its member. The initialization of this type can be done as usual.

```
struct employee = { ‘ Rajashekar’, 7777};
```

5.3.3 Arrays of Structures Contain Arrays

Arrays of structures can be defined and in that type of structure variables of array type can be used as members.

Example

```
struct rk
{
   int empno;
  char ename[20];
   flat  salary;
  } mark[50];
```

In the above, mark is an array of 50 elements and such element in the array is of structure type rk. The structure type rk, in turn contains ename as array type which is a member of the structure. Thus mark is an array of sutures and these structures in turn holds character names in array ename.

The initialization of the above type can be done as:

```
{
7777, ‘ Prasad’ , 56800.00}
};
i.e  mark[0] . empno = 7777;
    mark[0] . eame  = ‘Prasad’;
     mark[0] . salary =  56800.00
```

Program

Write a C program to accept the student name, rollno, average marks present in the class of student and to print the name of students whose average marks are greater than 40 by using structure concept with arrays.

```
# include <stdio.h>
main( )
{
 int  i, n,
 struct
 {
 char name [20];
```

```
  int rollno;
  flat avgmarks;
 }
class [40];
 printf (" Enter the no. of students in the class\n"0;
scanf ( " %d", & n );
for ( i = 0, i < n, i++)
{
    print ( " Enter students name, rollno, avgmarks\n");
    scanf ( " %s %d", &class[i].name, class[i].rollno, &class[i].avgmarks)'
}
printf (" The name of the students whose average");
printf ( " marks is greater than 40 \n");
for ( i = 0, i < n, i++)
if ( class[i].avgmarks > 40)
pirintf (" %s", class[i].name);
}
```

5.3.4 Advantages of Structure Type over Array Type Variables

1. Using structures, we can group items of different types within a single entity, which is not possible with arrays, as array stores similar elements.
2. The position of a particular structure type variable within a group is not needed in order to access it, whereas the position of an array member in the group is required, in order to refer to it.
3. In order to store the data about a particular entity such as a 'Book', using an array type, we need three arrays, one for storing the 'name', another for storing the 'price' and a third one for storing the 'number of pages' etc., hence, the overhead is high. This overhead can be reduced by using structure type variable.

4. Once a new structure has been defined, one or more variables can be declared to be of that type.

5. A structure type variable can be used as a normal variable for accepting the user's input, for displaying the output etc.,

6. The assignment of one 'struct' variable to another, reduces the burden of the programmer in filling the variable's fields again and again.

7. It is possible to initialize some or all fields of a structure variable at once, when it is declared.

8. Structure type allows the efficient insertion and deletion of elements but arrays cause the inefficiency.

9. For random array accessing, large hash tables are needed. Hence, large storage space and costs are required.

10. When structure variable is created, all of the member variables are created automatically and are grouped under the given variable's name.

5.4 Structure Contains Pointers

A pointer variable can also be used as a member in the structure.

Example:

```
struct
{
        int *p1;
        int  * p2;
} *rr;
```

In the above, *rr is a pointer variable of structure type which holds inside it another two pointer variables p1 and p2 as its members.

```
# include <stdio.h>
main( )
{
 sturct
    int  *p1, *p2;
} *rr;
```

```
int a, b ;
 a = 70;
 b = 100'
rr — p1 = &a;
rr — p2 = & b;
printf( " The contents of pointer variables");
printf( " Present in the structure as members are \n");
printf ( '%d %d", *rr — p1, *rr — p2);
}
```

In the above program, two pointer variables p1 and p2 are declared as members of the structure and their contents / variables are printer after assignment in the program.

5.4.1 Self Referential Structures

Structures can have members which are of the type the same structure itself in which they are included, This is possible with pointers and the phenomenon is called as self referential structures**.**

A self referential structure is a structure which includes a member as pointer to the present structure type.

The general format of self referential structure is

```
struct  parent
{
    memeber1;
    memeber2;
    ————————;
    ————————;
     struct  parent *name;
};
```

The structure of type parent is contains a member, which is pointing to another structure of the same type i.e. parent type and name refers to the name of the pointer variable.

Here, name is a pointer which points to a structure type and is also an element of the same structure.

Example

```
struct element
{
    char name{20};
   int  num;
    struct  element * value;
}
```

Element is of structure type variable. This structure contains three members

- a 20 elements character array called **name**
- An integer element called **num**
- a pointer to another structure which is same type called **value.** Hence it is self referential structure.

These structure are mainly used in applications where there is need to arrange data in ordered manner.

5.5 Unions

Introduction

A Union is a collection of heterogeneous elements. That is, it is a group of elements; each element is of different type. They are similar to structures. However, there is a difference in the way the structures members and union members are stored. Each member within a structure is assigned its own memory location. But the union members all share the common memory location. Thus, unions are used to save memory. Unions are chosen for applications involving multiple members, where values need to be assigned to all of the members at any one time.

5.6 Definition of Union

Union is a data type through which objects of different types and sizes can be stored at different times.

The general form of union type variable declaration is

Union name

```
{
data type  member-1;
        data type member-2;
data type  member-3;
        ……….
        ……….
data type member-n;
}
```

The declaration includes a key word Union to declare the union data type. It is followed by user defined name, followed by curly braces which includes the members of the union

Example

```
union       value
{
        int no;
        float sal;
        char sex;
};
```

Characteristics of Union

1. Union stores values of different types in a single location in memory
2. A union may contain one of many different types of values but only one is stored at a time.
3. The union only holds a value for one data type. If a new assignment is made the previous value has no validity.
4. Any number of union members can be present. But, union type variable takes the largest memory occupied by its members.

5.7 Differences between Structure and Unions

Structure	Union

1. Struct StructureName
{
datatype
member-1;
datatype
member-2;
….
….
datatype
member-n;
};
	1. Union name {
	datatype
	member-1;
	datatype
	member-2;
	….
	….
	datatype
	member-n;
	};

2. Every structure member is allocated memory when a structure variable is defined	2. The memory equivalent to the largest item is allocated commonly for all members
3. All the members can be assigned values at a time	3. Values assigned to one ember may cause the change in value of other members.
4. All members of a structure can be initialized at the same time	4. Only one union member can be initialized at a time
5. value assigned to one member will not cause the change in other members	5. Value assigned to one member may cause the change in value of other members.
6. The usage of structure is efficient when all members are actively used in the program	6. The usage o union is efficient when members of it are not required to be accessed at the same time.

Model Questions

Short Answer Type Questions - 2 Marks

1. What is Structure? Write the syntax of "structure" (struct).
2. What is Pointer? Which variables are used to represent it?
3. What are the advantages of pointers?
4. What are the operators used with pointers.

5.Write various operations performed by structure.

6 . What is a nested structure?

7. What advantage of structures over an Array?

8. What is Union? Write the syntax of Union.

Long Answer Type Questions - 6 Marks

1. What is Structure? Explain in detail.
2. Explain the advantages of structure type over the array the variable.
3. Explain about structure and arrays.
4. What is Union? Explain in detail.
5. What are differences between Structure and Unions?