UNIT 1

Introduction to Problem Solving Techniques

Structure

1.0 Introduction

1.1 Procedure (steps involved in problem solving)

1.2 Algorithm

1.3 Flow Chart

1.4 Symbols used in Flow Charts

1.5 Pseudo Code

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

2.0 Introduction

2.1 Character Set

2.2 Structure of a 'C' Program

2.3 Data Types in 'C'

2.4 Operations

2.5 Expressions

2.6 Assignment Statement

2.7 Conditional Statements

2.8 Structure for Looping Statements

2.9 Nested Looping Statements

2.10 Multi Branching Statement (Switch), Break and Continue

2.11 Differences between Break and Continue

2.12 Unconditional Branching (Go to Statement)

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");
else
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)
gotobig;
else
gotosmall;
big :printf("big value is %d",a);
gotostop;
small :printf("small value is %d",b);
gotostop;
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

3.0 Introduction

3.1 Functions

3.2 Differences between Function and Procedures

3.3 Advantages of Functions

3.4 Advanced features of Functions

3.5 Recursion

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

4.0 Introduction

4.1 Definition of Array

4.2 Types of Arrays

4.3 Two - Dimensional Array

4.4 Declare, initialize array of char type

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

5.0 Introduction

5.1 Definition of Structure

5.2 Structure Declaration

5.3 Structurs and Arrays

5.4 Structure contains Pointers

5.5 Unions

5.6 Definition of Union

5.7 Differences between Structure and Union

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?

Computer / Digital Computer

→ Computer
derived from "Compute" means Calculate.

→ It is an electronic device that process data & convert it into information that is useful to people.

→ It is a machine that performs pre-defined/ programmed/ control operations that are expressible in numerical or logical term

→ It works on binary information ie 0, 1

→ Every large information is breaked down into smaller units & each small unit is represented by binary codes

eg: Personal Computer

Operations of Computer

→ **Input:** Process of capturing or acquiring the information, or it is process of accepting data or information

→ **Process:** It is transformation process. It converts input data into useful information.

→ **Output:** It is the process by which the useful information is made available to user by printing or displaying on screen.

→ **Storing:** Digital computers are capable of storing information or instructions, so that user can retain or retrive it whenever required.

→ **Controlling:** It is the process of directing the manner & sequence in which all the operations are to be performed

Functional Components of Computer

Input Unit:

→ It act as communication medium between human & computer

→ It accepts input from the user & provide that data to the computer in the form which computer understands.

→ eg. Keyboard → used to enter text & numeric data

Mouse → used to position the screen & thereby enter the information by selecting option.

Output Unit:

→ It is also a communication medium between computer & human.

→ After the task of computer is finished, output unit sends result to user by using output device.

eg. Monitor → Screen that displays the result

Printer → used to get the information in printed form on paper.

① Arithmetic & logic Unit (ALU)

→ The ALU performs arithmetic operations like addition, subtraction & logic operations like OR, AND, invert etc

→ The data stored in memory unit is transferred to ALU. The ALU performs operation and result is stored in internal memory unit of CPU

② Control Unit (CU)

→ Control unit controls all the operations which internally takes place within the CPU & also the operations of CPU related to input/output devices

→ Control unit issues control instructions to the different parts of the computer which specify what action these parts have to perform.

③ Memory Unit (MU)

→ Memory unit stores the data read from input device & provide it for processing when required.

→ It also stores intermediate & final result of processing.

Computer System

- Hardware
- Software

Hardware: The physical part that make up a computer which are interconnected through electronic devices, & which can be seen and touched are called hardware.

ex: Keyboard, CPU, Monitor etc.

Software: Computer cannot think & perform operations of its own. Sequence of instructions are to be given to perform any operation.

→ The set of sequential instructions and data which performs the operation is called program.

The program or collection of such program is called Software.

Types of Hardware:

→ **Processor**: The execution of instructions that are supplied to the computer through input device is performed by 'Processor'. It also control, coordinate & carried out the execution related operations in computer.

→ **Input Devices**: The function of input devices is to apply data to the computer for processin

* Keyboard → It is used to enter text & numeric data
* Mouse → It is used to position the screen cursor.
* Trackball → It is used to produce screen cursor movement. It is 2 dimensional positioning device.
* Joystick → It has small vertical lever (called stick) mounted on base. It is also used in 2 dimensional positioning & movement. It is used to play computer games.
* Scanner: It is used to store drawing, graphs photos or text.

Types of Software:

→ **System Software:** System software is a set of programs which are required for controlling, integrating & managing the hardware components of computer system.

- It is needed for the basic functioning of computer system.
- System software interact with hardware or applications and allow user to work on computer system. (Interface between user & computer)
- eg → Operating System, Device Drivers, etc

→ **Application Software:** Computer system cannot perform different tasks just loading system software. Different tasks are performed by the help of different application softwares.

- The application software performs specific task assigned to it

- eg MS WORD for writing
  Calculator for calculations.

* <u>Microphone</u>: It allows user to input data in the form of voice.

→ <u>Output Devices</u>: The function of output device is to present processed data to user.

* Monitor: The computer sends processed data to monitor which displays the result to user

* Printer/Plotter → When output is needed on paper, the computer sends output to printer.

* Headphones or Speaker: When user needs output in the form of audio, the computer sends output to speakers.

PROGRAM :

Computer performs variety of tasks, but only when instructions are supplied to the computer.

⇒ When a set of sequential instructions is written to perform any task by computer it is called Computer Program.

- Computer executes the program, ie it follow the instructions & perform actions accordingly.

ALGORITHM :

Algorithm is finite sequence of steps required to solve the given problem

⇒ Same problem can be solved with different methods, so more than one algorithm may exists for the solution of one problem.

Properties

- Steps used in algorithm must be unambiguous and precisely defined
- Uncertainity about the instruction to be executed next should be avoided
- Steps in algorithm should be finite & should be terminated properly.

example

Algorithm to calculate addition of two Nos.

1. Start.
2. Read two numbers as A & B
3. Add numbers A & B and store result as C
4. Display C
5. Stop.

Flow Charts

A pictorial representation of an algorithm is called flow chart.

In flow chart the steps in the algorithm are represented in the form of different shapes of boxes & the logical flow is indicated by interconnecting arrows.

eg

Rules for Preparing Flow charts

→ Flow chart should be neat & easy to follow so that it will be clearly understood.

→ A logical start & end must be given to the flowchart.

→ Flow chart include necessary steps in logical order

→ Only two directions left to right or top to bottom are allowed for a procedure in flowchart.

→ There is only one flow line should come out from process symbol

→ The decision making symbol should have only one incoming flow line, however it may have two or three out-going flow lines

→ Terminal symbol ie: start & stop have only one flow-line.

Basic Symbols used in Flow Charts :

Symbol	Name	Description
↑↓→←	Flow Lines	Flow Lines connect symbols in Flow-Chart & indicates direction of flow.
	Terminal (Start/ Stop)	This is used to represent start & end of flow chart
	Input/output	It represents information which system reads as input or sends as output
	Processing	Processing of system is represented by this symbol eg: arithmatic operation, etc
	Decision	This symbol is used to check any condition or take decision for which there are two or more options (generally is/ls)

Q. Flow Chart & Algorithm to find sum & Average of 3 Numbers.

Algo

1. START
2. PRINT " Enter three number"
3. INPUT A, B, C
4. SUM = A+B+C
5. AVG = SUM/3
6. PRINT SUM, AVG
7. STOP

Flow Chart

Q. Algo & Flow Chart to print number divisible by 7, from 1 to N numbers

Algo

1. START
2. PRINT " Enter number N"
3. INPUT N
4. COUNT ← 1
5. WHILE COUNT ≤ N
   Begin
   Compute R = Count % 7
   If R = 0
   PRINT COUNT
   End If
   Compute COUNT ← COUNT+1
   END while
6. STOP

Q. Algo & Flow chart to find the sum of digits of a given number

Algo

1. START
2. PRINT " Enter Number"
3. INPUT N
4. SUM ← 0
5. If N=0
   PRINT SUM & goto step 6
   ELSE
   Q ← N/10
   R ← N%10
   SUM ← SUM+R
   N ← Q
   goto step 5
6. STOP

Q. Algorithm & Flow chart for Reverse digit of given number

Algo

1. START
2. PRINT "Enter Number"
3. INPUT N
4. SUM ← 0
5. If N = 0

   PRINT SUM & goto Step 6

   ELSE

   Q ← N/10

   R ← N%10

   SUM ← SUM×10 + R

   N ← Q goto step 5
6. STOP

* Fibonacci Series
* Factorial
* Prime Number
*

Eg: Algorithm & Flow chart to find largest of 3 numbers

Algo.

① START

② PRINT "Enter three numbers"

③ INPUT A, B, C.

④ If A > B Then

    If A > C

        PRINT A is largest

    Else

        PRINT C is largest

    End If

Else

    If B > C

        Print B is largest

    Else

        PRINT C is largest

    End If

End If

⑤ STOP

START
READ A B C
Is A > B ?
No
Yes
Is B > C ?
No
Is A > C
No
Yes
PRINT B is largest
PRINT C is largest
PRINT A is largest
STOP

Classification of Computers

Micro Computer :

→ A micro computer is small, low cost digital computer

→ It consists of microprocessor, storage unit, an input channel, & output channel.

→ In micro computer CPU is usually a single integrated circuit called microprocessor.

→ These types of computers are used in small industrial control, process control & where storage & speed requirement is moderate.

Desktop Computers :

→ A Desktop computer is the most common computer which usually have a seprate monitor and peripherals like keyboard and mouse attached to computer

→ Internal hardware consists of RAM, CPU, mother-board, graphic card etc.

Laptop Computer :

Laptop is computer in which all the peripherals are combined within the same small hand handeled structure.

Built in Keyboard, Touch pad replaces external Keyboard & mouse.

Tablet Computer

Tablet Computers shaped in the form of notebook or slate having capabilities of being writting on it.

User can use a stylus or digital pen & operate computer without having need of Keyboard or mouse.

The size of Tablets are very effective feature in the growth of Tablets.

Smart Phone

SmartPhone is an electronic device that integrate functionality of mobile phone.

Smart phone feature tends to include internet access, email access, built in camera, music-Player.

Smart phones ~~provide~~ are capable of performing almost all tasks that a normal computer can perform.

Mini Computer :

Mini Computer are evolved in 1960s describe the "small" things as compared to the previous generations.

Mini computers are of middle range computing capabilities in between mainframes & Micro Computer.

Mini Computers are best suited for Client-Server architecture where 4 to 200 computers are simultaneously access.

Work Station :

Work stations are computing desktop microcomputer designed primarly to be used by one person at a time.

→ These work stations are typically specialized featured & more power specialized computing than simple desktop PC.

Network Server :

→ Network server is a powerful computer System with special software that work as the centralized primary computer in network.

→ It manages & controls the computers of individual users connected in network.

Mainframe Computers

Mainframe computers are used for typically bulk data processing, like in banks, government universities.

Users can access the mainframe computers through a device called as "terminal"

The size of mainframe computers is very large & processing speed is very powerful

Super Computer:

→ Super computer is most powerful & physically one of the largest computer.

→ Super Computers are used for specific purposes like weather maps forcasting, Atom bomb formation, Satellite launch etc.

→ PARAM is super Computer made in India which has processing speed of upto 1 teraflops (1 trillion process per second).

RAM Vs ROM

⇒ * RAM stands for random access memory.

* Computer uses RAM to hold the program code & data during computation.

* All memory locations in RAM are accessed at almost the same speed.

* RAM is volatile memory ie the content of RAM is erased when computer is switched off.

* ROM stands for Read only memory.
* ROM contains critical computer programs such as booting program (that helps computer to start)
* Speed of data access in ROM will depend upon the location of data item.
* ROM is non-volatile memory ie the content of ROM is not erased when computer is switched off.

Operating System :

→ An operating system is a very-very large and complex program that controls and coordinates the use of hardware among the various system programs and application programs for various users.

→ It simply provide an environment within which other program can do useful work.

→ Every computer must have an operating system to run programs.

eg: Window, Linux, DOS etc.
Android.

- An operating system provides the interface between the hardware and user program.
- It is also responsible for security, ensuring that unauthorized user do not access the system
- When computer boots up (starts) operating system is the first program to run.

Functions of an Operating System

⇒ Process Management & Scheduling

Operating system allows different processes that are running to work in secured environment and also make resource available to processes Operating System also decides which process will execute & when.

⇒ Device Management

Operating system controls the peripherals that are attached to the computer. Process can access the devices attached to computer System by obtaining permission from Operating System.

⇒ File Management

Creation & deletion of files in computer is enabled by the help of operating system. Storing these files in computer's memory is also the work of Operating System.

Memory Management :

Operating system manages the memory of computer How & where the content is needed to be stored in memory.
It also organise the use of memory between programs.

→ Operating system also provides security to the users by assigning them password & acess rights.

→ Operating system also provide a facility for users to communicate with other users.

→ Operating system deals with different types of errors that may occur in computer.

Number System

Decimal Number System :

→ Any digit in decimal number system is represented in the form of ones, tens, hundreds, thousand - so on

→ eg. 5678.9 is represented as
$5000 + 600 + 70 + 8 + 0.9$

→ Any number in decimal number system is written as $(5678.9)_{10}$ where 10 subscript represents the base.

→ The leftmost digit is called most significant digit & rightmost digit is called least significant digit.

Binary Number System :

→ Binary system with its two digits is a base-two system

→ The two binary digits are 0 & 1

→ eg 10110101 is a binary number

→ Binary numbers are represented as $(10110101)_2$ where 2 is base.

Octal Number System :

→ The Octal number system use first eight digits of decimal number system ie 0, 1, 2, 3, 4, 5, 6 & 7

→ The base of digits in octal number system is 8

→ Any digit in Octal number system is represented as $(5632.471)_8$

Hexadecimal Number System :

→ The hexa decimal number system has a base of 16 having 16 characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E & F

→ An Example of hexa decimal number is 3FD.84

→ Hexadecimal number is represented as a base of 16 ie $(3FD.84)_{16}$

Binary	Decimal	Octal	Hexadecimal
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
~~1001~~ 1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

3.2. CONVERSION OF A BINARY NUMBER TO A DECIMAL NUMBER

The weight of nth digit of the number from the right hand side = nth digit × $(\text{Base})^{n-1}$.

If we have to convert a binary number to its decimal equivalent we take the following steps:

The weight of the nth bit of the number from the right hand side = nth bit × 2^{n-1}

In Binary System Base = 2

Example: Convert the binary number 1101 to its decimal equivalent

Binary number	1	0	0	1
		↓	↓	↓ ↓
Bit position of this binary number form right hand side	4th	3rd	2nd	1st

The weight of first bit $= 1 \times 2^0 = 1$

The weight of second bit $= 0 \times 2^1 = 0$

The weight of third bit $= 1 \times 2^2 = 4$

The weight of fourth bit $= 1 \times 2^3 = 8$

Decimal equivalent = total weight = $(13)_{10}$

3.3. CONVERSION OF A DECIMAL NUMBER TO A BINARY NUMBER

In a decimal number the 1st position from the right hand side is 1 for 1st, 2nd for 10s, 3rd for 100s, 4th for 1000s and so on. Similarly, in a binary number the 1st position from right hand side is for 1, 2nd for 2, 3rd for 4, 4th for 8, 5th for 16 and so on. This fact is utilized for the conversion of a decimal number to its binary equivalent. A method is to be developed to determine which multiples of 2 are

Conversion in Number System

Binary to Octal

- $(10101101.0111)_2$

421	421	421		421	421
010	101	101	.	011	100

2 5 5 . 3 4

$\Rightarrow (255.34)_8$

Binary to Hexa decimal

$\Rightarrow (110110110.1001101)_2$

8421	8421	8421		8421	8421
0011	0110	1110	.	1001	1010

3 6 E . 9 A

$\Rightarrow (36E.9A)_{16}$

Binary to Decimal

$$(\overset{7}{1}\overset{6}{0}\overset{5}{1}\overset{4}{0}\overset{3}{1}\overset{2}{1}\overset{1}{0}\overset{0}{1}.\overset{-1}{0}\overset{-2}{1}\overset{-3}{1}\overset{-4}{1})_2$$

$$1\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$

$$128 + 0 + 32 + 0 + 8 + 4 + 0 + 1$$

$$= 173.$$

.0111

$$0\times2^{-1}+1\times2^{-2}+1\times2^{-3}+1\times2^{-4}$$

$$= 0+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}$$

$$= 0.4375$$

$$\Rightarrow (173.4375)_{10}$$

Decimal to Binary

173.4375

2	173	
2	86	1
2	43	0
2	21	1
2	10	1
2	5	0
2	2	1
	1	0

$$(10101101.0111)_2$$

0 $\quad .4375\times2 = .875$

1 $\quad .875\times2 = 1.75$

1 $\quad .75\times2 = 1.5$

1 $\quad .5\times2 = 1.00$

Decimal to Octal

$(173.4375)_{10}$

8	173	
8	21	5
	2	5

$3 \leftarrow .4375 \times 8 = 3.500$

$4 \leftarrow .5 \times 8 = 4.000$

255

$\Rightarrow (255.34)_8$

Decimal to Hexa Decimal

$(173.4375)_{10}$

16	173	
	10 → A	D

$7 \leftarrow .4375 \times 16 = 7.00$

$(AD.7)_{16}$

Octal to Binary

$(255.34)_8$

2	5	5	.	3	4
421	421	421		421	421
010	101	101		011	100

$$(10101101.0111)_2$$

Octal to Decimal

$$(255.34)_8$$

$$2\times8^2 + 5\times8^1 + 5\times8^0$$

$$128 + 40 + 5$$

$$= 173$$

$$3\times8^{-1} + 4\times8^{-2}$$

$$= 3\times\frac{1}{8} + 4\times\frac{1}{64}$$

$$= .4375$$

$$(173.4375)_{10}$$

Octal to Hexadecimal

$$(255.34)_8$$

10101101 . 0111

8421	8421	.	8421
1010	1101	.	0111
A	D	.	7

$$(AD.7)_{16}$$

Hexadecimal to Binary

$$(AD.7)_{16}$$

8421	8421	.	8421
1010	1101	.	0111

$$(10101101.0111)_2$$

Hexadecimal to Decimal

$$(AD.7)_{16}$$

$$10 \times 16 + 13 \times 16^0$$

$$= (173.4375)_{10}$$

$$7 \times \frac{1}{16} = .4375$$

Hexadecimal to Octal

$(AD.7)_{16}$

8421 8421 . 8421
1010 1101 0111

1010 1101 . 0111

421 421 421 . 421 421
010 101 101 011 100

2 5 5 . 3 4

$(255.34)_8$

Examples

→ $(11011110.01101)_2 = (?)_{10}$ $(222.40625)_{10}$

→ $(489.287)_{10} = (?)_2$ $(111101001.01001001011)_2$

→ $(ABC.F2)_{16} = (?)_8$ $(5274.744)_8$

→ $(10001)_{10} = (?)_2$ $(100111000 10001)_2$

→ $(100)_{10} = (?)_8$ $(144)_8$

→ $(10101110 1010 1101.011010)_2 = (?)_{16} = (?)_8$ $(AEAD.68)_{16}$

→ $(623)_8 = (?)_2$ $(110 010 011)_2$ $(127255.32)_8$

→ $(1011.10)_2 = (?)$

COMPUTER LANGUAGES

- Programs should be written in computer languages called programming Languages
- The programming languages consist of a set of characters & symbols
- There are specified rules of writing programs to communicate with computers.
- There are syntax rules to write the program which should be followed to obtain desired result.

Types of Programming Languages

Machine Language :

The computer can understand only machine language which uses 0s & 1s. In machine language the different instructions to the computer are formed by taking different combinations of 0s & 1s.

This is lowest level of programming language in which instruction consists of two parts "opcode" & "operand"

OPCODE	OPERAND

The first part opcode (operation code) tells which operation is to be performed.
Second part operand tells the operation is to be performed on which data.

Advantages:

① Translation Free: Computer understands machine language so conversion is not required in case of machine language program.

② High Speed: As there is no need of translation Execution of machine language program is very fast.

Disadvantages:

① Complex language: It is very hard to develop a program using machine language

② Error Prone: The chances of occurance of error in machine level program is more as all coding is in the form of 0s & 1s.

③ Tedius: Writing program, making changes, corrections & modifications is complex task in Machine language.

④ Time consuming: It is very time-consuming to write entire code in 0 & 1.

Assembly language :

→ To make programming easier, assembly language is developed which is logically equivalent to machine language, but is easy to read, write & understand.

→ Assembly language is symbolic programming language that use symbolic notation to represent machine language instructions.

→ These languages are known as low level language

→ These languages uses mnemonic codes, which are easy to remember.

eg. ADD for addition
CMP for compare
MP for multiply
STO for store

Syntax

Label	Instruction (Opcode)	(operand)	Comment
START	MOV	A, B	copy B to A

* label is symbolic name used to point instruction
* Instruction specify opcode & operand
* Comment is non executable part, specify the action.

Assembler

→ Program written in assembly language require a translator to convert them into machine language

→ Assembler substitute the required machine code for each instruction in assembly language.

→ The program written in assembly language is called <u>source code</u> & after converting it into machine language, it is called <u>object code</u>.

→ Thus Assembler converts whole assembly language program into binary code & then it assemble the machine understandable code into main memory for execution.

Advantages of Assembly language :

1. They are easy to understand and use.
2. They are less error prone than machine language.
3. They require less resources than high level language hence more efficient.

Disadvantages

1. Machine Dependent : Depend on architecture of computer
2. Harder to learn : It is harder to learn as compared to high level languages
3. Development process is quite slow.
4. Less efficient than machine language
5.

High Level Language :

A programming language in which the program statement are not closely related to internal characterstics of computer is called high-level language

→ These languages are machine independent.

→ High level language fall somewhere close to general day to day language & makes programming ~~easy~~ easier & less error prone.

→ eg. C++, Java, Visual Basic etc.

Advantages :

→ Programs are easily readable & understandable

→ They are machine independent

→ Faster & easier ~~deve~~ development

→ Documentation is easy.

Disadvantages

→ Poor control on hardware

→ Less efficient

→ Porting program to new machine is not easy.

Compiler :

→ A compiler is a program which translates the source code written in high level language into computer readable machine language that can be directly loaded & executed.

→ Compiler replaces single high level statement with several machine language statements.

→ The compiler stores the entire high level program, scan it and translate the whole program into equivalent machine language program.

→ During the translation, compiler reads source program written in high level language. If any syntax or grammatical error is found, it generates an error message.

→ The machine code of high level language will not created untill all the errors are removed from source code.

→ * The compiler scans the whole program at once and report ~~the~~ all the errors at same time.

Interpreter

→ Interpreter is also a program which translates the source code of high level language into machine language

→ Unlike compiler, interpreter translates high-level Language statement into machine code & execute it immediately, before translating the next source Language statement.

→ When an error is found the execution of program is halted & an error message is displayed.

Linker :

→ A software consists of millions of lines of programming statements. It is usually much more effective to divide this code into logical group & store in different modules

→ Due to which each module is individual, tested and compiled & later joined together to form large executable program.

→ The single coherent program is formed by linking together the several modules by a system software called <u>Linker</u>.

→ Linker produced a linked file contains the binary codes for all the combined modules.

Loader :

→ Loader is a part of operating system that brings the executable file available on disk into memory & execute it. It performs the function of scheduling & execution

→ Loader also allocates the memory space for the programs

→ Loader also prepares the program to execute from its storage area

→ Loader also places instructions & data (machine code) into the memory

Unit-2

C Language :

C is a programing language developed at AT&T's Bell Laboratories of USA in 1972. It was designed & written by Dennis Ritchie.

C language is simple, more reliable, and easy to use.

In industry C language is still survived from last three decades.

Major part of popular ~~operation~~ operating system like windows, UNIX, Linux are still written in C.

The Character Set

Character denotes any alphabet, digit or special symbol used to represent information.

eg: Alphabets - A, B, - - - - - Y, Z
a, b, - - - - y, z

Digits - 0, 1, 2, 3 - - - - - 9

Special Symbol - ! @ # % & ( ) - _ { }
[ ] : ; " ' ? $

Constants, Variable & Keywords

→ A constant is an entity that doesn't change during the entire source code.

Rules for Integer Constants

(a) An integer constant must have at least one digit.

(b) It must not have a decimal point

(c) It can be either positive or negative.

(d) If no sign preceds an integer constant, it is assumed to be positive

No commas or blanks are allowed within integer constant.

(f) The allowable range for integer constant is -2147483648 to +2147483647. (for Visual Studio & gcc compiler)

for Turbo C (-32768 to 32767).

eg 426, +782, -8023

Real Constants:

Real constants are often called as floating point constants. Real constant could be written in two forms - Fractional & Exponential.

Rules

(a) A real constant must have at least one digit

(b) It must have a decimal point.

(c) It could be either positive or negative.

(d) Default sign is positive

(e) No comma or blank are allowed within a real constant.

eg: 325.34
426.0
-32.79

In exponential form of representation, the real constant is represented in two parts. Part appearing before 'e' is called mantissa whereas the part following 'e' is called exponent

Thus 0.000342 can be represented as 3.4e-4.

<u>Rules</u>:

(a) The mantissa part & the exponential part should be seprated by letter e or E.

(b) The mantissa part may have positive or negative sign.

(c) Default sign of mantissa is positive

(d) The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

(e) Range of Real constants expressed in exponential form is -3.4e38 to 3.4e38

eg +3.2e-5
4.1e8
-0.2E+3

Character Constants

* A Character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
  Both the inverted commas should point left.

* Maximum length of a character constant is 1

  eg 'A', '5' '='

Variables:

An entity that may vary during program execution is called variable.
Variable names are given to location of memory, These locations can contain integer, real or character constants.

In any language, the type of variables that it can support depend on the type of constant that it can handle.

For example an integer variable can hold only integer constant, Real variable can hold Real constant & so on.

Rules for Constructing Variables Name

(a) A variable name is any combination of 1 to 31 alphabets, digits or underscore.

(b) The first ~~alphabet~~ character in variable name is always a alphabet or underscore.

(c) No commas or blanks are allowed within variable name.

(d) No special symbol other than underscore (_) can be used in variable name.

eg: int a, si, m_1;
float rate;
char code

Keywords

→ Keywords are the words whose meaning has already been explained to the C compiler

→ These keywords cannot be used as variable name

→ Some time these keywords are also known as reserved keywords or "Reserved words"

There are 32 keywords reserved in C-language.
ie

Auto	double	int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
Continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Data Type

Data Type indicate the type of data that a variable can hold. The data may be numeric, alphabatic, symbolic etc.

In C language the data type of any data is necessarly known in order to use them in program.

There are broadly two types of data types in C

→ Built in data types (Pre existing that are known to compiler)

→ User defined data types (These are defined by the

Built in Data Types

They are basic or primitive data type ~~of data that a variable can hold~~ and designate one single value. There are four basic fundamental built in data type in C

* Integer (int) size is 2 bytes
* Floating point (float) size is 4 bytes
* Double (double) size is 8 bytes
* Character (char) size is 1 byte.

Integer:

This is a keyword used to indicate an integer number which is sequence of digits without a decimal point.

The range of integer values lies between -32,768 to 32,767

eg -247, 14020, 27246, + 1966

Invalid

3,333 (comma not allowed)
-34.0 (decimal point)
+ 3,486.30 (comma & decimal)
999999 (out of range)

Float :

This is a keyword used to indicate the floating point number. Floating point number are same as decimal number but they include fractional part as well.

eg -263.238
2.63238E+02 $\{2.63238 \times 10^{2}\}$
2.63E-02 $\{2.63 \times 10^{-2}\}$

Char :

This is a keyword used to indicate the character type data. A character in C language is defined as the single character enclosed within a pair of apostrophes.

eg 'a', 'p', '2', '?'

Double :

This is a keyword used to indicate a double precision floating number. Normally it is equivalent to float, but the number of digits stored after decimal are 16 in double as compared to 6 in float.

eg 234.0000

Data Type Modifier

The basic data types can be modified using a series of type modifiers to fit the requirements of a particular program.

Signed

This can be applied to integer variables. The default declaration assumes a signed number. The range of signed integer lies between -32768 to 32767

Unsigned

It can be used for integer, It can be combined with long or short. If the keyword unsigned is used before integer the range of positive integers are doubled & negative integers are removed.

Range is 0 to 65535

Long: This is also applied to int data types when applied to int, it essentially doubles the length in bits of data type that it modifies.

It can also be used with double.

Long int ranges from -2147483648 to 2147483647

Short: This makes the size of an int half.

Default declaration of int is short.

Range is same as that of intgers in C language

Range is -32768 to +32767

Variable Declaration

⇒ If the variable is declared as <u>const variable</u>, its value cannot be changed. An attempt to modify its value will result in an error message.

eg: Const float PI = 3.142;

⇒ If the variable is defined as <u>volatile variable</u> or just as variable then its value can be modified at any time.

eg Volatile int x;

⇒ multiple assignment to the variables is possible in C language, It is possible to assign a value to multiple variables at same time (called as multiple assignment)

eg: int a = b = c = 20;
float x = y = z = 3.687;

Backslash Constant

→ A backslash constant is a combination of two characters in which the first character is always a backslash (\), and the second character can be any one of characters. a, b, n, t etc.

→ The backslash character constant are also called escape sequence.

→ The backslash constants are used in output statements.

eg

"\a"	System bell
"\b"	back space
"\n"	new line
"\t"	horizontal tab
"\v"	vertical tab
"\0"	null (end of string)

Symbolic Constants :

A symbolic Constant is a name that substitute a numeric constant, a character constant or string Constant.

eg

```
area = 3.142 * radius * radius;
```

here it can be defined as

```
# define PI 3.142
```

now. the formula is modified as

```
area = PI * radius * radius;
```

⇒ Generally, the symbolic constants are written in upper case letter.

⇒ Symbolic constant must be defined at the beginning of program. ie before main ( ).

Storage Class :

The storage class determines the part of the memory where the variable would be stored & how long the storage allocation continues to exist ie how long the variable would exist.

It also determines the scope of the variable which specifies the part of program where the variable name is visible.

Storage class also determines the initial value of the variable, if the variable is not explicitly initialized by any value.

① Automatic Storage Class

→ Variables declared at the start of a block are automatic variables

→ memory is allocated automatically upon entry to the block & freed automatically upon exiting the block

→ Variables are accessible only within the block, hence the scope of automatic variables are local to the block in which they are defined.

→ No block outside the defining block may have direct ~~acc~~ access to automatic variables.

→ By default all variables declared within a function are automatic

They can be declared as:

```
int i;
auto int i;
```

→ To declare any variable as automatic, a keyword auto is prefixed before its ~~name~~ declaration.

No Need

eg

```
# include <stdio.h>
void main ( )
{
    autoblock ( );
    autoblock ( );
}

auto block ( )
{
    auto int x = 10;
    printf ("\nvalue of x is %d", x);
    x = x+10;
}
```

Output is

Value of x is 10
Value of x is 10.

Note:

(i) Storage : Memory

(ii) Default initial value : Garbage value.

(iii) Scope : Local to block in which it is declared

(iv) Life : As long as program control remains within the block in which it is declared.

eg

```
# include <stdio.h>
void main ( )
{
auto int i, j ;
printf ("%d %d \n", i, j);
}
```

Output

1211 221 {garbage value}

eg

```
# include <stdio.h>
void main( )
{
    auto int i= 1;
    {
        auto int i=2;
        {
            auto int i=3;
            printf ("%d", i);
        }
        printf ("%d", i);
    }
    printf ("%d", i)
}
```

Output:

3 2 1

Register Storage Class

→ The register storage class specifier indicates to the compiler that the value of variable should reside in computer system register.

→ As the size of computer memory is limited, it is possible to keep only few variables, if compiler is not able to put variables in computer register, then the

variable are treated as automatic variables.

→ Thus it is not sure that the variable is stored in computer register.

→ Frequently used variables like loop counter are stored in registers.

Note :

Storage : CPU registers

Default Initial value: Garbage Value

Scope: local to block in which variable is declared

Life: Till control remains within the block in which the variable is defined

eg

```
# include <stdio.h>
void main ( )
  {
    register int i;
      for (i=1; i<=10; i++)
          printf ("%d", i);
  }
```

Static Storage Class

→ The static storage class provides a life time to the variables over the entire program, but it restricts the scope of static variable.

→ These variables are declared using the static keyword.

→ Default initial value of static variables is zero

→ Static variables retain their values even after the block in which they are defined terminates.

→ The value of a static variable in a function is retained between repeated call to the same function.

Note

Storage : Memory
Default initial value: Zero
Scope : Local to the block in which the variable is declared

Lifetime : Variables retains its value between different function calls.

eg

```
# include < stdio.h >
void main ( )
{
  increment ( );
  increment ( );
  increment ( );
}

increment ( )
{
  static int i = 1
  printf ("%d \n", i);
  i = i+1;
}
```

Output

In case of (auto int).	In case of (static int)
1	1
1	2
1	3

External Storage Class :

→ External storage class variables are accessible through out the program execution ie all the functions/ blocks use that particular variable.

→ These variables are known as global variables.

→ External variables are declared outside any functional block

→ Memory is allocated to these variables when program begins execution & the lifetime is untill the program terminates

→ Default initial value of these variables is zero.

→ Scope of these variables is global ie entire program code has accessibility to these variables

→ All functions may access external variable by its name, but if any variable declared within the function have same name, reference to the name access the local variables.

Note:

Storage: Memory
Default initial value: Zero
Scope: In all functions of program (global)
Lifetime: Throughout program execution

eg

```
# include < stdio.h>
int i;
void main()
  { printf ("\n i= %d", i);
    increment ( );
    increment ( );
    decrement ( );
    decrement ( );
  }
increment ( )
  { i= i+1;
  } printf ("\n on incremental i= %d", i);
decrement ( )
  { i= i-1;
  } printf ("\n on decrement i= %d", i);
```

output

```
i=0
on incremental i= 1
on incremental i=2
on decrement i= 1
```

Operators

→ In order to carry out basic arithmatic operations & logical operations operators are needed.

→ Operators act as connectors and they indicate what type of operation is being carried out.

→ The values that can be operated by these operator are called operands.

→ operators in C language are broadly classified in three categories.

Unary Operator

An operator that acts upon only one operand is known as unary operator. Some unary operators are:

① Unary minus

② Logical NOT operator (discussed in Logical operator)

③ Bitwise Complementation (discussed in Bitwise operator)

Unary Minus:

Any positive operand associated with unary minus operator gets its value changed to negative.

eg $a = 3, \quad b = 4$

$$c = a + (-b)$$
$$= 3 + (-4).$$
$$= -1$$

Binary Operator:

These operators acts upon two operands, hence named as binary. The binary operator are further divided in four categories

① Arithmatic Operator

② Logical Operator

③ Relational Operator

④ Bitwise operator

Arithmatic Operator:

Arithmatic operators performs basic arithmetic operations such as addition, substraction, multiplication & division.

Modulus operator is also used in C language that finds the remainder after integer division.

Operation	Operator	Precedence	Associativity
Addition	+	2	Left to Right
Substraction	−	2	Left to Right
multiplication	*	1	Left to Right
Division	/	1	Left to Right
Modulus	%	1	Left to Right

* The precedence indicates which terms to be evaluated.

* Associativity indicates the order in which the terms are evaluated.

Evaluation of Arithmatic Expression :

(i) a+b

⇒ There is only one operator (+), hence the value of 'a' is added to value of 'b'.

(ii) x*y/z

⇒ There are two operators ('*' & '/') having same priority. Now expression is scanned from left to right.

I^st x*y is evaluated

IInd The result of (x*y) is divided by z,

(iii) a+b*c/d-e

I^st b*c

IInd (b*c)/d

IIIrd a+((b*c)/d)

IVth (a+((b*c)/d))-e

eg:

```
int n1, n2;
n1 = 5;
n2 = 3;
```

(i) $n_1 + n_2 = 5 + 3 = 8$

(ii) $n_1 - n_2 = 5 - 3 = 2$

(iii) $n_1 * n_2 = 5 * 3 = 15$

(iv) $n_1 / n_2 = 5/3 = 1$

(v) $n_1 \% n_2 = 5 \% 3 = 2$

eg

```
float a, b;
a = 5.0
b = 2.0
```

(i) $a + b = 5.0 + 2.0 = 7.0$

(ii) $a - b = 5.0 - 2.0 = 3.0$

(iii) $a * b = 5.0 * 2.0 = 10.0$

(iv) $a / b = 5.0 / 2.0 = 2.5$

<u>Note:</u> % operator cannot be used with floating values

Rules for Evaluation of Arithmetic Expressions

① If the expression have paranthesses, then the expression inside parentheses must be evaluated first. (Rest priority & associativity will remain same)

② If unary minus is present in the expression then the term associated with unary minus must be evaluated before other expressions.

eg

$2 * ((i/3) + 4 * (j-2))$ where $i=8$ & $j=5$

Thus

$$2 * ((8/3) + 4 * (5-2)) = 2 * (2 + 4 \times 3)$$
$$= 2 * (2 + 12)$$
$$= 2 * 14$$
$$= \boxed{28}$$

Program to illustrates Basic Arithmetic Operators


```
# include <stdio.h>
# include <conio.h>
void main ( )
  {
    int a, b;
    int sum, dif, prod, quot, rem;
    printf (" Enter value of first number");
    scanf ("%d", &a);
    printf (" Enter value of second number");
    scanf ("%d", &b);

    sum= a+b;
    dif= a-b;
    prod = a*b;
    quot = a/b;
    rem = a%b;
    printf ("\n Sum = %d", sum);
    printf ("\n Difference=%d", dif);
    printf ("\n PRODUCT = %d", prod);
    printf ("\n Quotient = %d", quot);
    printf ("\n Remainder=%d", rem);
  }
```

Increment & Decrement Operator

* Increment operator is used to increment the value of an integer quantity by one.

→ This is represented by '++' symbol, This symbol can be placed before or after the integer variable.

ie

```
int a;
    a++ or ++a;   {both are allowed}
```

* Decrement operator is used to reduce the value of an integer quantity by one.

→ This is represented by '--' symbol. This symbol can be placed before or after integer variable.

ie

```
int b
    b-- or --b  {both are allowed}
```

Difference between Pre & Post increment/decrement

++a → indicates preincrement in the value of variable before it is used

a++ → indicates post increment in the value of variable after it is used

eg

```
# include <stdio.h>
void main( )
{
    int a=3, b=6;
    printf ("a = %d \n", a++);
    printf ("b = %d \n", ++b);
}
```

Output

a = 3
b = 7

Similarly

--b → decrement the value of variable first then use it

b-- → use the value of variable then decrement it

```
# include <stdio.h>
void main( )
{
  int m=6, n=6;
  printf("m=%d", --m);
  printf("\n n=%d", n--);
}
```

<u>output</u>

```
m = 5
n = 8
```

Relational operators

Relational operators are used to compare two operands, They also define the relationship existing between two constants or variables.

→ They result in either a TRUE or FALSE value. The value of TRUE is nonzero (ie 1) and the value of FALSE is zero

→ There are six relational operators in C language.

Operator	Meaning	Precedence	Associativity
<	is less than	1	Left to Right
<=	lesser than or equal to	1	Left to Right
>	greater than	1	Left to Right
>=	greater than or equal	1	Left to Right
==	is equal to	2	Left to Right
!=	not equal to	2	Left to Right

eg

```
# include <stdio.h>
void main ()
{
    int a = 8;
    int b = -5;
printf("%d < %d is %d\n", a, b, a<b);
printf("%d == %d is %d\n", a, b, a==b);
printf("%d != %d is %d\n", a, b, a!=b);
printf("%d > %d is %d\n", a, b, a>b);
printf("%d <= %d is %d\n", a, b, a<=b);
printf("%d >= %d is %d\n", a, b, a>=b);
}
```

Output

```
8 < -5 is 0
8 == -5 is 0
8 != -5 is 1
8 > -5 is 1
8 <= -5 is 0
8 >= -5 is 1
```

Logical Operator

Logical operators are used to take decisions, There are three such operators, (AND, OR & NOT).

→ AND, OR are binary operator & NOT is unary operator

→ The result of these operators is either TRUE (1) or FALSE (0).

→ Logical operator are used to connect one or more relational expression.

Operator	Meaning	Precedence	Associativity
&& (double ampersand)	Logical AND	2	Left to Right
|| (double pipeline)	Logical OR	3	Left to Right
! (exclamation)	Logical NOT	1	~~Left to Right~~ Right to Left

AND

Operand 1	Operand 2	operand 1 && operand2
0	0	0
0	1	0
1	0	0
1	1	1

OR

Operand 1	Operand2	operand 1 ~~&&~~ || Operand2
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Operand	! Operand
0	1
1	0

eg Suppose

$a=2, \quad b=4, \quad c=3$

```
a&&b||c && (!b)
2&&4||3 && (!4)
2&&4||3&&0
1||3&&0
1||0
=1
```

Prog

```
# include <stdio.h>
void main( )
{
    int a=4, b=3, c;
    c= a&&b;
    b= a||b||c;
    a= a&&b||c;
    printf ("%d.%d %d", a, b, c);
}
```

Output

1, 1, 1

Bitwise Operator :

All the data stored in the computer's memory as a sequence of bits (ie 0, 1), some application needs manipulation of these bits.

→ To perform bitwise operations, C provides six operators.

→ These operators works on char & int variables but not on floating type data.

~~There~~ .

Operator	Symbol Name	Meaning
&	Ampersand	Bitwise AND
|	Pipeline	Bitwise OR
^	Caret	Bitwise XOR
~	Tide	1's complement (NOT)
<<	Double less than	Left shift of bits
>>	Double greater than	Right shift of bits

XOR

operand 1	Operand2	operand1 ^ operand2
0	0	0
0	1	1
1	0	1
1	1	0

eg int a=4, b=3;

For 8 bit Computer

Binary Equivalent of 4 & 3 are

a= 0000 0100

b= 0000 0011

now

a & b = 0000 0000

a | b = 0000 0111

a ^ b = 00000111

Bitwise Complement :

Let a=10

then a = 1010

if b = ~a = ~(1010)

then b = 0101

Bitwise Left shifting

In left or right shift operator case, the general form of representation is

<variable>	<shift operator>	<nob>

variable → int or char data element
shift operator → left shift or right shift operator
nob → number of bits to shift

eg: (i) a >> 1 or (ii) a << 2

Left shift

Let a = 132
or 10000100

Before left shifting

1	0	0	0	0	1	0	0

After one bit shifted to left

0	0	0	0	1	0	0	0

Right Shift

Before shifting

1	0	0	0	0	1	0	0

After one bit Right shifting

0	1	0	0	0	0	1	0

Program

```
# include < stdio.h >
Void main ( )
  {
   int x, y;
   x=128, y= 32;
  x = x>>1;
printf ("After right shifting by 1, x= %d \n", x);
  y = y<<2;
printf ("After Left shifting by 2, y= %d \n", y);
  }
```

Output

After right shifting by 1, x = 64

After left shifting by 2, y = 128

Ternary Operator

* Ternary operator is conditional operator in C.
* It takes three operands.
* '?' is used as a ternary operator in C
* The general form of a ternary expression is as follows:

<expression> ? <value1> : <value2>

where,

expression → Relational expression
value1 → value to be assigned on true
value2 → value to be assigned when the result of expression is false

eg: c = a<b ? a : b;

here c is assigned a value of a if a is less than b. Otherwise it will be assigned the value of b.

Prog

```
# include < stdio.h>
void main ()
{
    int a=4, b=5, result1, result2;
    result1 = a>b ? a : b;
printf ("The result1 = %d \n", result1);
```

```
result2 = a<b? a:b;
    printf ("The result 2= %d\n", result2);
}
```

<u>Output</u>:

The result 1 = 5

The result 2 = 4

Special Operator of C

<u>Comma Operator</u>: The comma operator is basically associated with for statement. It is also used to link two or more related expressions together. In such case the expression are evaluated in left to right fashion and we can obtain the value of the combined expression ~~from the value of the combined expression~~ from the value of rightmost expression.

eg. Sum = (a=12, b=22, a+b).

here there are 3 expressions, the value of last expression is set to sum.

Size of ( ) operator

→ The size of() operator returns the size (in number of bytes) of the operand.

→ It is written in lower case letters.

→ It must precede its operand.

→ The operand may be constant, variable or data type.

eg

```
x = size of (int);
y = size of (sum);
```

Program

```
void main()
{ int x = 10;
  float y = 6.5;
  Char ch = 'y';
printf("size of x = %d\n", size of (x));
printf("size of y = %d\n", size of (y));
printf("size of ch = %d\n", size of (ch));
printf("size of double = %d\n", size of (double))
}
```

Output

size of x = 2

size of y = 4

size of ch = 1

size of double = 8

Precedence & Associativity of all C operators

Operator Category	Operator	Precedence	Associativity
Parentheses, braces	(), []	1	Left to Right
Unary operator	+, -, ++, --, !, ~, &	2	Right to Left
multiplicative operator	*, /, %	3	Left to Right
Additive Operators	+, -	4	Left to Right
Shift operator	<<, >>	5	Left to Right
Relational operator	<, <=, >, >=	6	Left to Right
Equality operators	==, !=	7	Left to Right
Bitwise operators	&, ^, |	8	Left to Right
Logical operators	&&, ||	9	Left to Right
Conditional operator	?, :	10	Right to left
Assignment operator	=, +=, -=, *=	11	Right to left
Comma operator	,	1	Left Right

Assignment Operator:

→ Assignment operator assigns the value of expression on the right side of it to the variable on the left of it.

→ The assignment expression evaluates the operand on the right side of operator (=) & places its value in the variable on the left.

eg

```
a=4;
area= length * breadth;
```

C language also provides a compact way of writing assignment statements in expressions.

The general form of shorthand assignment in C is as follows:

Var	operator	=	expression	;

Var → Variable

operator → an arithmetic or bitwise operator

expression → is simply an expression.

Operator	Assignment (a & b)	Shorthand assignment
+	a = a + b	a += b
−	a = a − b	a − = b
*	a = a * b	a * = b
/	a = a / b	a / = b
%	a = a % b	a % = b

Data Type Conversion :

Some time it is needed to change the datatype of variable. When we change the nature of data stored in variable. This process is known as data type conversion or type casting.

Syntax (datatype) variable

⇒ eg

```
int K= 2;
float term;
term = 1/K ;
```

Here the value assigned to the term is 'zero' as of integer division.

But the real value which should be 0.5
25

for 0.5 datatype conversion is required which can be achieved as follows:

```
term = 1/(float)K;
```

Mixed Mode Operations :

⇒ Expressions that involves, elements of type real & integer are called mixed mode expressions

→ If an expression containing both integer & float values, then the result will be of type float.

→ If the result value is assigned to integer then the fractional part of float must be truncated.

→ Operators that can be used in mixed mode expressions are (+, -, *, /, %)

eg:

int a = 4, b = 2;

float x = 2, y = 3

(i) a = (x/y) + (a/b);

(ii) b = (a*x)/y;

(iii) x = a - b;

Solⁿ

(i)

$$a = (x/y) + (a/b)$$
$$= (2.0/3.0) + (4/2)$$
$$= 0.6 + 2$$
$$= 2.6$$

$$\boxed{a = 2}$$

(ii)

$$b = (a*x)/y$$
$$4 \times 2.0$$
$$= 8.0/3$$
$$= 2.66$$

$$\boxed{b = 2}$$

(iii)

$$x = a - b$$
$$= 4 - 2$$
$$= 2$$

$$\boxed{x = 2.0}$$

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

18CSS101J – Programming for Problem Solving Unit I

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

COURSE LEARNING RATIONALE (CLR)	*The purpose of learning this course is to:*
CLR -1:	Think and evolve a logically to construct an algorithm into a flowchart and a pseudocode that can be programmed
CLR -2:	Utilize the logical operators and expressions to solve problems in engineering and real-time
CLR -3:	Store and retrieve data in a single and multidimensional array
CLR -4:	Utilize custom designed functions that can be used to perform tasks and can be repeatedly used in any application
CLR -5:	Create storage constructs using structure and unions. Create and Utilize files to store and retrieve information
CLR -6:	Create a logical mindset to solve various engineering applications using programming constructs in C

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

COURSE LEARNING OUTCOMES (CLO)	*At the end of this course, learners will be able to:*
CLO - 1:	Identify methods to solve a problem through computer programming. List the basic data types and variables in C
CLO - 2:	Apply the logic operators and expressions. Use loop constructs and recursion. Use array to store and retrieve data
CLO - 3:	Analyze programs that need storage and form single and multi-dimensional arrays. Use preprocessor constructs in C
CLO - 4:	Create user defined functions for mathematical and other logical operations. Use pointer to address memory and data
CLO - 5:	Create structures and unions to represent data constructs. Use files to store and retrieve data

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

LEARNING RESOURCES	
S. No	**TEXT BOOKS**
1.	*Zed A Shaw, Learn C the HardWay: Practical Exercises on the Computational SubjectsYou Keep Avoiding (Like C),AddisonWesley, 2015*
2.	*W. Kernighan, Dennis M. Ritchie,The C Programming Language, 2nd ed. Prentice Hall, 1996*
3.	*Bharat Kinariwala,Tep Dobry, Programming in C, eBook*
4.	*http://www.c4learn.com/learn-c-programming-language/*

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY,
DELHI-NCR CAMPUS

UNIT I INTRODUCTION

Evolution of Programming & Languages - Problem Solving through Programming - Creating Algorithms - Drawing Flowcharts - Writing Pseudocode - Evolution of C language, its usage history - Input and output functions: Printf and scanf - Variables and identifiers – Expressions - Single line and multiline comments - Constants, Keywords - Values, Names, Scope, Binding, Storage Classes - Numeric Data types: integer -

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY,
DELHI-NCR CAMPUS

UNIT I INTRODUCTION

floating point - Non-Numeric Data types: char and string - Increment and decrement operator - Comma, Arrow and Assignment operator - Bitwise and Sizeof operator

1. 1 Evolution of Programming & Languages

- A Computer needs to be given instructions in a programming language that it understands
- Programming Language
  - Artificial language that controls the behavior of computer
  - Defined through the use of syntactic and semantic rules
  - Used to facilitate communication about the task of organizing and manipulating information
  - Used to express algorithms precisely

Evolution of programming languages

1. 1 Evolution of Programming & Languages Contd

Period	Programming Langugaes
1950's	Creation of high-level languages
1960's	Forth. Simula I. Lisp, Cobol
1970's	Pascal, C language
1980's	ML. Smalltalk, C++
1990's	Java, Perl, Python languages
2000	Internet Programming
2010	Concurrency and asynchronicity. JavaScript and Go language

1. 2 Problem Solving through Programming

❑***Problem*** - Defined as any question, something involving doubt, uncertainty, difficulty, situation whose solution is not immediately obvious

❑***Computer Problem Solving***

- ❑ Understand and apply logic
- ❑ Success in solving any problem is only possible after we have made the effort to understand the problem at hand
- ❑ Extract from the problem statement a set of precisely defined

1. 2 Problem Solving through Programming Contd…

i.Creative Thinking

- Proven method for approaching a challenge or opportunity in an imaginative way
- Process for innovation that helps explore and reframe the problems faced, come up with new, innovative responses and solutions and then take action
- It is generative, nonjudgmental and expansive
- Thinking creatively, a lists of new ideas are generated

1. 2 Problem Solving through Programming Contd…

ii.Critical Thinking

- Engages a diverse range of intellectual skills and activities that are concerned with evaluating information, our assumptions and our thinking processes in a disciplined way so that we can think and assess information more comprehensively
- It is Analytical, Judgmental and Selective
- Thinking critically allows a programmer in making choices

1. 2 Problem Solving through Programming Contd…

1. 2 Problem Solving through Programming Contd…

- ***Program -*** Set of instructions that instructs the computer to do a task
- ***Programming Process***
  - *a) Defining* the Problem
  - *b) Planning* the Solution
  - *c) Coding* the Program
  - *d) Testing* the Program
  - *e) Documenting* the Program

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

1. 2 Problem Solving through Programming Contd…

1. 2 Problem Solving through Programming Contd…

- A typical programming task can be divided into two phases:

i.Problem solving phase

- Produce an ordered sequence of steps that describe solution of problem this sequence of steps is called an ***Algorithm***

ii.Implementation phase

- Implement the program in some programming language

- ***Steps in Problem Solving***

a) Produce a general algorithm (one can use ***pseudocode***)

1. 2 Problem Solving through Programming Contd…

b) Refine the algorithm successively to get step by step detailed ***algorithm*** that is very close to a computer language

c) ***Pseudocode*** is an artificial and informal language that helps programmers develop algorithms

- ❑ Pseudocode is very similar to everyday English

1. 3 Creating Algorithms

- An informal definition of an algorithm is:

i

Algorithm: a step-by-step method for solving a problem or doing a task.

1. 3 Creating Algorithms Contd…

- What are Algorithms for?
  - A way to communicate about your problem/solution with others
  - A possible way to solve a given problem
  - A "formalization" of a method, that will be proved
  - A mandatory first step before implementing a solution
- ***Algorithm Definition*** - "A finite sequence of unambiguous, executable steps or instructions, which, if followed would ultimately terminate and give the solution of the problem"

1. 3 Creating Algorithms

- **Notations**
  - Starting point
  - Step Numbers – Positions in Algorithm
  - Incoming Information - Input
  - Control Flow – Order of evaluating Instructions
  - Statements
  - Outgoing Information - Output
  - Ending Point

1. 3 Creating Algorithms Contd…

- ***Properties of an algorithm***
  - **Finite**: The algorithm must eventually terminate
  - **Complete**: Always give a solution when one exists
  - **Correct (sound)**: Always give a correct solution
- ***Rules of Writing an Algorithm***
  - Be consistent
  - Have well Defined input and output
  - Do not use any syntax of any specific programming language

1. 3 Creating Algorithms Contd…

- Algorithm development process consists of five major steps
  - **Step 1**: Obtain a description of the problem
  - **Step 2:** Analyze the problem
  - **Step 3:** Develop a high-level algorithm
  - **Step 4:** Refine the algorithm by adding more detail
  - **Step 5:** Review the algorithm

1. 3 Creating Algorithms Contd…

Example

❑*Problem*

a) Develop an algorithm for finding the largest integer among a list of positive integers

b) The algorithm should find the largest integer among a list of any values

c) The algorithm should be general and not depend on the number of integers

1. 3 Creating Algorithms Contd…

❑*Solution*

a) To solve this problem, we need an intuitive approach

b) First use a small number of integers (for example, five), then extend the solution to any number of integers

c) The algorithm receives a list of five integers as input and gives the largest integer as output

(12 8 13 9 11) Input data
Largest 12 12 8 13 9 11 List
Step 1
Largest 12 12 8 13 9 11 List
Step 2
Largest 13 12 8 13 9 11 List
Step 3
Largest 13 12 8 13 9 11 List
Step 4
Largest 13 12 8 13 9 11 List
Step 5
FindLargest
(13) Output data

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

(12 8 13 9 11) **Input data**

Set Largest to the first number.

Step 1

If the second number is greater than Largest, set Largest to the second number.

Step 2

If the third number is greater than Largest, set Largest to the third number.

Step 3

If the fourth number is greater than Largest, set Largest to the fourth number.

Step 4

If the fifth number is greater than Largest, set Largest to the fifth number.

Step 5

FindLargest

(13) **Output data**

(12 8 13 9 11) **Input data**

↓

Step 0 | Set Largest to – ∞

Step 1 | If the current number is greater than Largest, set Largest to the current number.

⋮

Step 5 | If the current number is greater than Largest, set Largest to the current number.

FindLargest

↓

(13) **Output data**

Input data (*n* integers)
Set Largest to – ∞
Repeat the following step *n* times:
If the current integer is greater than Largest, set Largest to the current integer.
FindLargest
Largest

1. 3 Creating Algorithms Contd…

Example 2: **Print 1 to 20**

- **Step 1:** Start
- **Step 2:** Initialize X as 0,
- **Step 3:** Increment X by 1,
- **Step 4:** Print X,
- **Step 5:** If X is less than 20 then go back to step 2.
- **Step 6:** Stop

1.3 Creating Algorithms Contd…

Example 3

Convert Temperature from Fahrenheit (°F) to Celsius (°C)

- **Step 1:** Start
- **Step 2:** Read temperature in Fahrenheit
- **Step 3:** Calculate temperature with formula C=5/9*(F-32)
- **Step 4:** Print C
- **Step 5:** Stop

1.3 Creating Algorithms Contd…

Example 4

Algorithm to Add Two Numbers Entered by User

- **Step 1:** Start
- **Step2:** Declare variables num1, num2 and sum.
- **Step 3:** Read values num1 and num2.
- **Step 4:** Add num1 and num2 and assign the result to sum.

  sum←num1+num2
- **Step 5:** Display sum Step 6: Stop

1. 3 Creating Algorithms Contd…

- **Write an Algorithm to:**

1) Find the Largest among three different numbers
2) Find the roots of a Quadratic Equation
3) Find the Factorial of a Number
4) Check whether a number entered is Prime or not
5) Find the Fibonacci Series

1. 3 Drawing Flowcharts

- Diagrammatic representation
- Illustrates sequence of operations to be performed
- Each step represented by a different symbol
  - Each Symbol contains short description of the Process
- Symbols linked together by arrows
- Easy to understand diagrams
- Clear Documentation
- Helps clarify the understanding of the process

Flowchart Symbols

Flowcharts are used to illustrate algorithms in order to aid in the visualisation of a program.

Flowcharts are to be read top to bottom and left to right in order to follow an algorithms logic from start to finish. Below is an outline of symbols used in flowcharts.

Terminator
Used to represent the Start and end of a program with the Keywords BEGIN and END.

Decision
Used to split the flowchart sequence into multiple paths in order to represent SELECTION and REPETITION.

Process
An instruction that is to be carried out by the program.

Input / Output
Used to represent data entry by a user or the display of data by the program.

Arrow
Indicates the flow of the algorithm pathways.

Subprogram
References another program within the program.

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS

1. 3 Drawing Flowcharts Contd…

1. 3 Drawing Flowcharts Contd…

❑Guidelines for Preparing Flowchart

- ❑Logical order of requirements
- ❑Ensure that Flowchart has logical *Start* and *Stop*
- ❑Direction is from Top to bottom
- ❑Only one flow line is used with Terminal Symbol
- ❑Only one flow line should come out of a Process symbol
- ❑Only one flow line should enter a Decision symbol but multiple lines may leave the Decision symbol

1. 3 Drawing Flowcharts Contd…

- **Guidelines for Preparing Flowchart Contd…**
  - Write briefly within Symbols
  - Use connectors to reduce number of flow lines
  - Avoid intersection of flow lines
  - Test Flowchart through simple test data
  - Clear, Neat and easy to follow

1. 3 Drawing Flowcharts Contd…

www.cprogrammingcodes.blogspot.com

start
Read n
N=5
Set i=1
count=0
i <=n
no
yes
count
==2
yes
Display n
is prime
Display
not prime
STOP
n% i==0
yes
no
count=count+1
i=i+1

1. 4 Writing Pseudocode

- Pseudo – Imitation / False
- Code – Instructions
- **Goal:** To provide a high level description of the Algorithm
- **Benefit:** Enables programmer to concentrate on Algorithm
- Similar to programming code
- Description of the Algorithm
- No specific Programming language notations
- Pseudo Code transformed into actual program code

1. 4 Writing Pseudocode Contd…

a) Guidelines for Writing Pseudo Code

- **Write only one Statement per line**
  - Example – Pseudo Code for calculating Salary

1. **READ** name, hourly rate, hours worked, deduction rate
2. Gross pay = hourly rate * hours worked
3. deduction = gross pay * deduction rate
4. net pay = gross pay – deduction
5. **WRITE** name, gross, deduction, net pay

1. 4 Writing Pseudocode Contd…

b) Capitalize Initial Keyword

- Keywords to be written in capital letters
- Examples: **READ, WRITE, IF, ELSE, WHILE, REPEAT, PRINT**

c) Indent to show Hierarchy

- Indentation shows the structure boundaries
- Sequence
- Selection
- Looping

1.4 Writing Pseudocode Contd…

d) End Multiline structures

- Each structure must end properly
- Example: IF statement must end with ENDIF

e) Keep Statements Language independent

- Resist the urge to write Pseudo Code in any programming language

1. 4 Writing Pseudocode Contd…

- **Advantages**
  - Easily typed in a Word document
  - Easily modified
  - Simple to Use and understand
  - Implements Structured Concepts
  - No special symbols are used
  - No specific syntax is used
  - Easy to translate into Program

1. 4 Writing Pseudocode Contd…

- **Disadvantages**
  - No accepted Standard
  - Cannot be compiled and executed

1. 4 Writing Pseudocode Contd…

❑Write an Pseudo Code to:

1) Add three numbers and Display the result
2) Calculate Sum and product of two numbers
3) Input examination marks and award grades according to the following criteria:
   a) > = 80 Distinction
   b) > = 60 First Class
   c) > = 50 Second Class
   d) < 40 Fail

1. 4 Writing Pseudocode Contd…

1. Pseudo Code to Add Three Numbers

- Use Variables: sum, num1, num2, num3 of type integer
- ACCEPT num1,num2,num3
- Sum = num1+num2+num3
- Print sum
- End Program

1. 4 Writing Pseudocode Contd…

2. Calculate Sum and product of two numbers

- Use Variables: sum, product, num1, num2 of type real
- DISPLAY "Input two Numbers"
- ACCEPT num1,num2
- Sum = num1+num2
- Print "The sum is", sum
- product = num1*num2
- Print "The product is", product
- End Program

1. 4 Writing Pseudocode Contd…

3. Input examination marks and award grades

- Use Variables: mark of type integer
- If mark >=80 DISPLAY "Distinction"
- If mark >=60 and mark <80 DISPLAY "First Class"
- If mark >=50 and mark <60 DISPLAY "Second Class"
- If mark <50 DISPLAY "Fail"
- End Program

1. 5 History & Evolution of C

- C– General Purpose Programming Language
- Developed by Dennis Ritchie in 1972
- Developed at Bell Laboratories
- Principles taken from BCPL and CPL
- Structured Programming Language
- C Program
  - Collection of Functions
  - Supported by Clibrary

1. 5 History & Evolution of C Cont…

Father of C Programming : Dennis Ritchie

Born On	September 9 1941
Born in	Bronxville – New York
Full Name	Dennis MacAlistair Ritchie
Nickname	DMR
Nationality	American
Graduate From	Harvard University
Graduate In	Physics and Applied Mathematics
Webpage	http://cm.bell-labs.com/who/dmr/
Dead On	October 12 2011

1. 5 History & Evolution of C Cont…

1960	Algol	• International Group
1967	BCPL	• Martin Richards
1970	B	• Ken Thomson
1972	Traditional C	• Dennis Ritchie
1978	K&R C	• kernighan & Ritchie
1989	ANSI C	• ANSI Commitee
1990	ANSI/ISO C	• ISO Commitee
1999	C99	• Standerd Commitee

Evolution of C

1. 5 History & Evolution of C Cont…

- ***Why the Name "C" was given ?***
  - Many of C's principles and ideas were derived from the earlier language B
  - BCPL and CPL are the earlier ancestors of B Language (CPL is common Programming Language)
  - In 1967, BCPL Language (Basic CPL) was created as a scaled down version of CPL
  - As many of the **features were derived from "B" Language the new language was named as "C"**.

1. 5 History & Evolution of C Cont…

- **Characteristics of 'C'**
  - Low Level Language Support
  - Structured Programming
  - Extensive use of Functions
  - Efficient use of Pointers
  - Compactness
  - Program Portability
  - Loose Typing

1. 5 History & Evolution of C Cont…

- **Advantages of C**
  - Compiler based Language
  - Programming – Easy &Fast
  - Powerful and Efficient
  - Portable
  - Supports Graphics
  - Supports large number of Operators
  - Used to Implement Datastructures

1. 5 History & Evolution of C Cont…

- **Disadvantages of C**
  - Not a strongly typed Language
  - Use of Same operator for multiple purposes
  - Not Object Oriented

1. 6 Structure of 'C' Program

- Structure based on Set of rules defined by the Compiler
- **Sections**

1) Documentation
2) Preprocessor
3) Global Declaration
4) main( ) function
5) Local Declaration
6) Program Statements

1. 6 Structure of 'C' Program Contd…

❑ Rules for Writing a C Program

a) All statements should be written in lower case

b) All statements should end with a semicolon

c) Upper case letters are used for symbolic constants

d) Blank spaces can be inserted between words

e) No blank space while declaring a variable, keyword, constant

f) Can write one or more statement in same line separated by comma

g) Opening and closing of braces should be balanced

1. 6 Structure of 'C' Program Contd…

```
/* Program to Find Area of Circle */
```
Comment

```
#include <stdio.h>
#include <conio.h>
```
Preprocessor Directives

```
const float pi = 3.14;
```
Global Declaration

```
void main( )
{
```
main Function

```
float area;  int r;
```
Local Declaration & Initialization

```
printf("Enter the Radius of the Circle");
scanf("%d", &r);
area = pi * r * r;
printf("The area of the Circle is %f", area);
getch( );
}
```
Execution

1. 6 Structure of 'C' Program Contd…

1) Documentation Section

- Used for providing Comments
- Comment treated as a single white space by Compiler
- Ignored at time of Execution: Not Executable
- Comment: Sequence of Characters given between /* and */
- **Example:** Program Name, Statement description

/* Program to Find Area of a Circle*/

1. 6 Structure of 'C' Program Contd…

2) Preprocessor Section

- Also called as Preprocessor Directive
- Also called as Header Files
- Not a part of Compiler
- Separate step in Compilation Process
- Instructs Compiler to do required Preprocessing
- Begins with # symbol
- Preprocessor written within < >

1. 6 Structure of 'C' Program Contd…

- **Examples**
  - #include <stdio.h>
  - #include <conio.h>
  - #include <math.h>
  - #include <stdlib.h>
  - #define PI 3.1412

1. 6 Structure of 'C' Program Contd…

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.

1. 6 Structure of 'C' Program Contd…

Directive	Description
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

1. 6 Structure of 'C' Program Contd…

3) Global Declaration Section

- Used to Declare Global variable (or) Public variable
- Variables are declared outside all functions
- Variables can be accessed by all functions in the program
- Same variable used my more than one function

1. 6 Structure of 'C' Program Contd…

4) main( ) Section

- main( ) written in all small letters (No Capital Letters)
- Execution starts with a Opening Brace : {
- Divided into two sections: Declaration & Execution
  - **Declaration :** Declare Variables
  - **Executable:** Statements within the Braces
- Execution ends with a Closing Brace : }
- ***Note:*** main( ) does not end with a semicolon

1. 6 Structure of 'C' Program Contd…

5) Local Declaration Section

- Variables declared within the main( ) program
- These variables are called Local Variables
- Variables initialized with basic data types

1. 8 C Programming Fundamentals

Steps in Learning English Language

Steps in Learning C

1. 8 C Programming Fundamentals Contd…

Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY,
DELHI-NCR CAMPUS

1. 8 C Programming Fundamentals

- **C Token** - Smalles
- C program broken ir
- Building Blocks of C

1. 9 Keywords

- **Keywords** – Conveys special meaning to Compiler
- Cannot be used as variable names

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

1. 10 Constants

- Definition :Value does not change during execution
- Can be a Number (or) a Letter
- **Types**
  - Integer Constants
  - Real Constants
  - Character Constant
    - Single Character Constants
    - String Constants

1. 10 Constants Contd…

1. 11 Variables & Identifiers

❑*Identifier*

- ❑ A string of alphanumeric characters that begins with an alphabetic character or an underscore character
- ❑ There are 63 alphanumeric characters, i.e., 53 alphabetic characters and 10 digits (i.e., 0-9)
- ❑ Used to represent various programming elements such as variables, functions, arrays, structures, unions
- ❑ The underscore character is considered

1.11 Variables & Identifiers Contd…

- **Rules for Identifers**
  - Combination of alphabets, digits (or) underscore
  - First character should be a Alphabet
  - No special characters other than underscore can be used
  - No comma / spaces allowed within variable name
  - A variable name cannot be a keyword
  - Variable names are case sensitive
- ***Variable Definition*** : Value changes during execution
- Identifier for a memory location where data is stored

1.11 Variables & Identifiers Contd…

- Variable name length cannot be more than 31 characters
- **Examples:** AVERAGE, height, a, b, sum, mark_1, gross_pay

Variable Declaration

- A variable must be declared before it is used
- Declaration consists of a data type followed by one or more variable names separated by commas.
- Syntax

datatype variablename;

1.11 Variables & Identifiers Contd…

- Examples

int a, b, c, sum; float avg;

char name;

Variable Initialization

- Assigning a value to the declared variable
- Values assigned during declaration / after declaration

1.11 Variables & Identifiers Contd…

- Examples

  i. *int a, b, c;*

     a=10, b=20, c=30;

  ii. *int a=10 ,b=10, c=10;*

- **Scope of Variables**
- Local Variables
- Global Variables

1. 12 Scope of Variables

❑*Definition*

- ❑ A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed

❑**Variable Scope** is a region in a program where a variable is declared and used

❑The ***scope*** of a variable is the range of program statements that can access that variable

❑A variable is ***visible*** within its scope and ***invisible*** outside it

1. 12 Scope of Variables Contd…

- There are three places where variables can be declared
  a) Inside a function or a block which is called **local** variables
  b) Outside of all functions which is called **global** variables
  c) In the definition of function parameters which are called **formal** parameters

1. 12 Scope of Variables Contd…

a)Local Variables

- Variables that are declared inside a function or block are called local variables
- They can be used only by statements that are inside that function or block of code
- Local variables are created when the control reaches the block or function containing the local variables and then they get destroyed after that
- Local variables are not known to functions outside their own

/* Program for Demonstrating Local Variables*/

```
#include <stdio.h>
int main ( )
{
        /* local variable declaration */
int a, b;  int c;
/* actual initialization */
a = 10; b = 20;
c = a + b;
printf ("value of a = %d, b = %d and c = %d\n", a,
b, c);
return 0;
}
```

1. 12 Scope of Variables Contd…

b) Global Variables

- Defined outside a function, usually on top of the program
- Hold their values throughout the lifetime of the program
- Can be accessed inside any of the functions defined for the program
- Can be accessed by any function
  - That is, a global variable is available for use throughout the entire program after its declaration

/* Program for Demonstrating Global Variables*/

```
#include <stdio.h>
                    /* global variable declaration */
int g;
int main ()
{
                    /* local variable declaration */
    int a, b;
                    /*  actual initialization */
    a = 10; b = 20;
    g = a +b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

1. 12 Scope of Variables Contd…

❑***Note:*** A program can have same name for local and global variables but the value of local variable inside a function will take preference

1. 12 Datatypes

- Defines a variable before use
- Specifies the type of data to be stored in variables
- Basic Data Types – 4 Classes
  a) int – Signed or unsigned number
  b) float – Signed or unsigned number having Decimal Point
  c) double – Double Precision Floating point number
  d) char – A Character in the character Set
- Qualifiers

1. 12 Datatypes Contd…

Variable Type	Keyword	Bytes Required	Range	Format
Character (signed)	Char	1	-128 to +127	%c
Integer (signed)	Int	2	-32768 to +32767	%d
Float (signed)	Float	4	-3.4e38 to +3.4e38	%f
Double	Double	8	-1.7e308 to + 1.7e308	%lf
Long integer (signed)	Long	4	2,147,483,648 to 2,147,438,647	%ld
Character (unsigned)	Unsigned char	1	0 to 255	%c
Integer (unsigned)	Unsigned int	2	0 to 65535	%u
Unsigned long integer	unsigned long	4	0 to 4,294,967,295	%lu
Long double	Long double	10	-1.7e932 to +1.7e932	%Lf

1. 12 Datatypes Contd…

a) Integer Data Type

- Whole numbers with a range
- No fractional parts
- Integer variable holds integer values only
- **Keyword:** int
- **Memory:** 2 Bytes (16 bits) or 4 Bytes (32 bits)
- **Qualifiers:** Signed, unsigned, short, long
- **Examples:** 34012, 0, -2457

1. 12 Datatypes Contd…

b) Floating Point Data Type

- Numbers having Fractional part
- Float provides precision of 6 digits
- Integer variable holds integer values only
- **Keyword:** float
- **Memory:** 4 Bytes (32 bits)
- **Examples:** 5.6, 0.375, 3.14756

1. 12 Datatypes Contd…

c) Double Data Type

- Also handles floating point numbers
- Double provides precision of 14 digits
- Integer variable holds integer values only
- **Keyword:** float
- **Memory:** 8 Bytes (64 bits) or 10 Bytes (80 bits)
- **Qualifiers:** long, short

1. 12 Datatypes Contd…

d) Character Data Type

- handles one character at a time
- **Keyword: char**
- **Memory:** 1 Byte (8 bits)

1. 13 Expressions

- **Expression :** An Expression is a collection of operators and operands that represents a specific value
- **Operator** : A symbol which performs tasks like arithmetic operations, logical operations and conditional operations
- **Operands** : The values on which the operators perform the task
- Expression Types in C
    a) Infix Expression
    b) Postfix Expression
    c) Prefix Expression

1. 13 Expressions Contd…

a) *Infix Expression*

- The operator is used between operands
- ***General Structure :*** Operand1 Operator Operand2
- ***Example :*** a + b

b) *Postfix Expression*

- Operator is used after operands
- ***General Structure :*** Operand1 Operand2 Operator
- ***Example :*** ab+

1. 13 Expressions Contd…

c) *Prefix Expression*

- Operator is used before operands
- ***General Structure :*** Operator Operand1 Operand2
- ***Example :*** +ab

1. 14 Input and Output Functions

- Ability to Communicate with Users during execution
- **Input Operation**
  - Feeding data into program
  - Data Transfer from Input device to Memory
- **Output Operation**
  - Getting result from Program
  - Data Transfer from Memory to Output device
- Header File : #include**<stdio.h>**

1. 14 Input and Output Functions Contd…

- **Input / Output Function Types**

  a) Formatted Input / Output Statements

  b) Unformatted Input / Output Statements

Console Input / Output Functions

Formatted Functions

Type	Input	Output
Char	scanf( )	printf( )
Int	scanf( )	printf( )
Float	scanf( )	printf( )
String	scanf( )	printf( )

Unformatted Functions

Type	Input	Output
char	getch( ) getche( ) getchar( )	putch( ) putchar( )
int	-	-
float	-	-
string	gets( )	puts( )

1. 14 Input and Output Functions Contd…

a) Formatted Input / Output Statements

- Reads and writes all types of data values
- Arranges data in particular format
- Requires **Format Specifier** to identify Data type
- Basic Format Specifiers
  - %d – Integer
  - %f – Float
  - %c – Character
  - %s - String

1. 14 Input and Output Functions Contd…

i. The scanf ( ) Function

- Reads all types of input data
- Assignment of value to variable during Runtime

- **Syntax**

scanf("Control String/Format Specifier", &arg1,&arg2,… &argn)

- Control String / Format Specifier
- arg1, arg2.,,, arg n – Arguments (Variables)
- &-Address

1. 14 Input and Output Functions Contd…

/* Giving Direct Input in Program */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int a;  a=10;
}
```

/*Getting Input using scanf ( ) function */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int a;
scanf("%d", &a);
}
```

1. 14 Input and Output Functions Contd…

```
/* Getting Multiple Input using
        scanf ( ) function */
#include<stdio.h>
        #include<conio.h>    void
        main( )
{
int a, b, c;
scanf("%d%d%d",&a,&b,&c);
}
```

```
/* Getting Multiple Different Inputs
using scanf ( ) function */
#include<stdio.h>
#include<conio.h>  void main( )
{
int a, b;  float c;
scanf("%d%d%f",&a,&b,&c);



}
```

1. 14 Input and Output Functions Contd…

```
/* Getting Multiple Input using scanf ( ) function */
#include<stdio.h>  #include<conio.h>  void main( )
{
int a, b;  float c;
scanf("%d %d", &a, &b);
scanf("%f ", &c);
}
```

1. 14 Input and Output Functions Contd…

ii. The printf ( ) Function

- To print Instructions / Output onto the Screen
- Requires Format Specifiers & Variable names to print data

- **Syntax**

printf("Control String/Format Specifier",arg1,arg2,… argn)

- Control String / Format Specifier
- arg1, arg2.,,, arg n – Arguments (Variables)

1. 14 Input and Output Functions Contd…

```
/* Example 1 – Using printf ( ) & scanf ( ) function */
#include<stdio.h>  #include<conio.h>  void main( )
{
int a;
printf("Enter the Value of a");  scanf("%d", &a);
printf("Value of a is %d", a);  getch( );
}
```

1. 14 Input and Output Functions Contd…

```
/* Example 2 – Using printf ( ) & scanf ( ) function */
#include<stdio.h>  #include<conio.h>  void main( )
{
int a, b, c;
printf("Enter the Value of a, b & c");  scanf("%d %d
%d", &a, &b, &c);
printf("Value of a, b & c is %d%d%d", a, b, c);  getch (
);
}
```

```
/* Example 3 – Using printf ( ) & scanf ( ) function */
#include<stdio.h>
#include<conio.h>  void main( )
{
int a, b;  float c;
printf("Enter the Value of a & b");  scanf("%d %d", &a,
&b);
printf("Enter the Value of a & b");
scanf("%f ", &c);
printf("Value of a, b is %d%d", a, b);  printf("Value of c
is %f", c);
getch ( );
}
```

1. 14 Input and Output Functions Contd…

```
/* Example 4 – Using printf ( ) & scanf ( ) function */
#include<stdio.h>  #include<conio.h>  void main( )
{
int a, b;  float c;
printf("Enter the Value of a, b & c");
scanf("%d %d%f", &a, &b, &c);
printf("Value of a, b & c is %d%d%f", a, b, c);  getch (
);
}
```

1. 14 Input and Output Functions Contd…

- **Try it Out Yourself ! Write a C program to:**

1) Add two numbers
2) To Multiply two floating point numbers
3) To compute Quotient and Remainder
4) To Swap two numbers

1. 14 Input and Output Functions Contd…

b) Unformatted Input / Output Statements

- Works only with Character Data type
- No need of Format Specifier
- **Unformatted Input Statements**

  i. **getch ( )** – Reads alphanumeric characters from Keyboard

  ii. **getchar ( )** – Reads one character at a time till enter key is pressed

1. 14 Input and Output Functions Contd…

iii. **gets ( )** – Accepts any string from Keyboard until Enter Key is pressed

- **Unformatted Output Statements**

i. **putch ( )** – Writes alphanumeric characters to Monitor (Output Device)

ii. **putchar ( )** – Prints one character at a time

iii. **puts ( )** – Prints a String to Monitor (Output Device)

1. 16 Single Line and Multiline Comments

Comment – Definition

- Used to provide information about lines of code
- Provide clarity to the C source code
- Allows others to better understand what the code was intended to
- Helps in debugging the code
- Important in large projects containing hundreds or thousands of lines of source code
- Types – Single line and multiline comment

1. 16 Single Line and Multiline Comments Contd…

a) Single Line Comment

- Represented by double slash\\

```
#include<stdio.h>
int main( ){
//printing information  printf("Hello C");
return 0;
}
```

1. 16 Single Line and Multiline Comments Contd…

b) Multi-Line Comment

- Represented by slash asterisk * … *\

```
#include<stdio.h>
int main( ){
/*printing information
Multi Line Comment*/  printf("Hello
C");
return 0;
}
```

1. 16 Single Line and Multiline Comments Contd…

Single-Line Comments	Multi-Line Comment
Starts with /* and ends with */	Starts with //
All Words and Statements written between /* and */ are ignored	Statements after the symbol // upto the end of line are ignored
Comment ends when */ Occures	Comment Ends whenever ENTER is Pressed and New Line Starts
e.g /* Program for Factorial */	e.g // Program for Fibonacci

1.15 Operators in C

- C supports rich set of built in Operators
- Used to manipulate Constants (Data) & Variables
- Part of Mathematical (or) Logical expressions
- Operators vs Operands
- **Operator – Definition**
  - Symbol (or) Special character that instructs the compiler to perform mathematical (or) Logical operations

1.15 Operators in C Contd…

❑ Classification of Operators

a) Increment & Decrement Operators

b) Comma Operator

c) Arrow Operator

d) Assignment Operators

e) Bitwise Operators

f) Sizeof Operator

1. 15 Operators in C Contd…

a)Increment and Decrement Operators

- Increment and decrement operators are unary operators that add or subtract one from their operand
- C languages feature two versions (pre- and post-) of each operator
  - Operator placed before variable (Pre)
  - Operator placed before variable (Post)
- The increment operator is written as ++ and the decrement operator is written as

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

- **Classification**
  - Pre Increment Operator
  - Post Increment Operator
  - Pre Decrement Operator
  - Post Decrement Operator

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

- ***Syntax***

(pre)++variable_name;	*(pre)- -variable_name;*

(Or)

variable_name++ (post);	*variable_name –(Post);*

- ***Examples***
  - ++count, ++a, ++i, ++count
  - Count++, a++, i++, count++

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

S. No	Operator type	Operator	Description
1	Pre Increment	++i	Value of i is incremented before assigning it to variable i.
2	Post Increment	i++	Value of i is incremented after assigning it to variable i.
3	Pre Decrement	-- i	Value of i is decremented before assigning it to variable i.
4	Post Decrement	i --	Value of i is decremented after assigning it to variable i.

```
/* Program for Post Increment*/
#include<stdio.h>
#include<conio.h>  void main( )
{
int i = 1;
while (i++<5)
{
printf("%d", i);

}
}
getch ( );
}
```

Output

1 2 3 4 5

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

- **Step 1 :** In this program, value of i "0" is compared with 5 in while expression.
- **Step 2 :** Then, value of "i" is incremented from 0 to 1 using post-increment operator.
- **Step 3 :** Then, this incremented value "1" is assigned to the variable "i".
- Above 3 steps are continued until while expression becomes false and output is displayed as "1 2 3

```
/* Program for Pre Increment*/
#include<stdio.h>
#include<conio.h>  void main( )
{
int i = 1;
while (++i<5)
{
printf("%d", i );

}
}
getch ( );
}
```

Output

1 2 3 4

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

- **Step 1 :** In above program, value of "i" is incremented from 0 to 1 using pre-increment operator.
- **Step 2 :** This incremented value "1" is compared with 5 in while expression.
- **Step 3 :** Then, this incremented value "1" is assigned to the variable "i".
- Above 3 steps are continued until while expression becomes false and output is displayed as "1 2 3 4".

```
/* Program for Post Decrement*/
#include<stdio.h>
#include<conio.h>  void main( )
{
int i = 1;
while (i--<5)
{
printf("%d", i );

}
}
getch ( );
}
```

Output

9 8 7 6 5

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

- **Step 1 :** In this program, value of i "10" is compared with 5 in while expression.
- **Step 2 :** Then, value of "i" is decremented from 10 to 9 using post-decrement operator.
- **Step 3 :** Then, this decremented value "9" is assigned to the variable "i".
- Above 3 steps are continued until while expression becomes false and output is displayed as "9 8 7

```
/* Program for Pre Decrement*/
#include<stdio.h>
#include<conio.h>  void main( )
{
int i = 1;
while (--i<5)
{
printf("%d", i);

}
}
getch ( );
}
```

Output

9 8 7 6

1. 15 Operators in C Contd…

a) Increment and Decrement Operators Contd…

- **Step 1 :** In above program, value of "i" is decremented from 10 to 9 using pre-decrement operator.
- **Step 2 :** This decremented value "9" is compared with 5 in while expression.
- **Step 3 :** Then, this decremented value "9" is assigned to the variable "i".
- Above 3 steps are continued until while expression becomes false and output is displayed as "9 8 7 6".

1. 15 Operators in C Contd…

b)Comma Operator

- Special operator which separates the declaration of multiple variables
- Has Lowest Precedence i.e it is having lowest priority so it is evaluated at last
- Returns the value of the rightmost operand when multiple comma operators are used inside an expression
- Acts as Operator in an Expression and as a Separator while Declaring Variables

1. 15 Operators in C Contd…

b) Comma Operator Contd…

```
#include<stdio.h>
int main( )
{
int i, j;  i=(j=10, j+20);
printf("i = %d\n j = %d\n" , i,j );
return 0;
}
```

1. 15 Operators in C Contd…

c)Arrow Operator (->)

- Arrow operator is used to access the structure members when we use pointer variable to access it
- When pointer to a structure is used then arrow operator is used

1. 15 Operators in C Contd…

d) Assignment Operators

- Assigns result of expression to a variable
- Performs Arithmetic and Assignment operations
- Commonly used Assignment operator: =
- ***Syntax***

variable = expression;

- ***Examples***
  - num = 25; age = 18; pi = 31.4; area = 3.14 * r * r;

1. 15 Operators in C Contd…

- Shorthand Assignment Operators

Simple Assignment Operator	Shorthand Operator
a = a + 1	a+=1
a = a – 1	a-=1
a = a * 2	a*=2
a = a / b	a/=b
a = a % b	a%=b
c = c * (a + b)	c *= (a + b)
b = b / (a + b)	b /=(a + b)

```
/* Program for Assignment Operations*/
#include<stdio.h>          #include<conio.h>
void main( )
{
                    int a;
a = 11;
a+ = 4;
printf("Value of A is %d\n",a);  a = 11;
a- = 4;
printf("Value of A is %d\n",a);  a = 11;
a* = 4;
printf("Value of A is %d\n",a);  a = 11;
a/ = 4;
```

```
printf("Value of A is %d\n",a);
a = 11;
a% = 4;
printf("Value of A is %d\n",a);  getch ( );
}
```

Output

Value of A is 15 Value of A is 7

Value of A is 44 Value of A is 2

Value of A is 3

THANKYOU

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

18CSS101J –Programming for Problem Solving

Unit II

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

UNIT II

Relational and logical Operators - Condition Operators, Operator Precedence - Expressions with pre / post increment Operator - Expression with conditional and assignment operators - If statement in expression - L value and R value in expression - Control Statements – if and else - else if and nested if, switch case - Iterations, Conditional and Unconditional Branching - For loop - While loop - do while, goto, break, continue –Array - Initialization and Declaration-

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

UNIT II

- Initialization: one Dimensional Array-Accessing, Indexing one Dimensional Array Operations - Array Programs – 1D

2.1 Operators in C

a) Relational Operators

b) Logical Operators

c) Conditional Operators

2.1 Operators in C Contd…

a) Relational Operators

- Binary Operators (or) Boolean Operators
- Produces an integer result
  - **Condition True :** Integer value is 1
  - **Condition False :** Integer value is 0
- Compares
  - Values between two variables
  - Values between variables and constants

2.1 Operators in C Contd…

a) Relational Operators Contd…

- **Relational Expression / Boolean Expression :** An expression containing a relational operator

Relational Operatiors

Operatior	Operations	Example
<	Less than	a<b
>	Greater than	a>b
<=	Less than or equal to	a<=b
>=	Greater than equal to	a>=b
=	Equal to	a==b
!=	Not equal to	a!=b

2.1 Operators in C Contd...

a) Relational Operators Contd...

- Consider a = 10 and b = 4. The relational expression returns the following integer values

Relational Expression	Result	Return Values
a < b	False	0
a > b	True	1
a < = b	False	0
a > = b	True	1
a = = b	False	0
a ! = b	True	1

/* Program for Relational Operations*/

```
#include<stdio.h>
int main( )
{
    int a,b;
    printf("Enter the two Values\n");
    scanf("%d%d", &a, &b);
    printf("a>b is %d\n", (a>b));
    printf("a<b is %d\n", (a<b));
    printf("a>=b is %d\n", (a>=b));
    printf("a<=b is %d\n", (a<=b));
    printf("a==b is %d\n", (a==b));
    printf("a!=b is %d\n", (a!=b));
    return 0;
}
```

Output

```
4
2
a > b is 1
a < b is 0
a > = b is 1
a < = b is 0
a = = b is 0
a ! = b is 1
```

2.1 Operators in C Contd…

b) Logical Operators

- Combines two or more relations
- Used for testing one or more conditions

SYMBOLS	MEANINGS	
"&&"	LOGICAL AND:	true when all expression are T false otherwise F
"||"	LOGICAL OR:	true when either expression is T false when both are F
"!"	NOT:	negation (T → F) and vice-versa

2.1 Operators in C Contd...

b) Logical Operators Contd...

- **Logical Expression / Compound Relational Expression :** An expression which combines two or more relational expression

Op1	Op2	Op1 && Op2	Op1 || Op2
F (0)	F (0)	F (0)	F (0)
F (0)	T (1)	F (0)	T (1)
T (1)	F (0)	F (0)	T (1)
T (1)	T (1)	T (1)	T (1)

2.1 Operators in C Contd...

b) Logical Operators Contd...

- Consider a = 10 and b = 4. The Logical expression returns the following integer values

Relational Expression	Result	Return Values
a < 5 && b > 2	True	1
a < 5 && b < 2	False	0
a > 5 && b < 2	False	0
a > 5 || b < 2	True	1
a < 5 || b < 2	False	0
a > 5 || b < 2	True	1

/* Program for Logical Operations*/

```
#include<stdio.h>
int main( )
{
    int age,height;
    printf("Enter Age of Candidate:\n");
    scanf("%d", &age);
    printf("Enter Height of Candidate:\n");
    scanf("%d", &height);
    if ((age>=18) && (height>=5))
        printf("The Candidate is Selected");
    else
        printf("Sorry, Candidate not Selected");
    return 0;
}
```

Output 1

Enter Age of Candidate: 18

Enter Height of Candidate: 6

The Candidate is Selected

Output 2

Enter Age of Candidate: 19

Enter Height of Candidate: 4

Sorry, Candidate not Selected

2.1 Operators in C Contd…

c) Conditional Operators

- ? and are the Conditional Operators
- Also called as Ternary Operators
- Shorter form of if-then-else statement
- ***Syntax***

Expression 1 ?Expression 2 : expression 3

- If expression 1 is true then the value returned will be expression 2
- Otherwise the value returned will be expression 3

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

```
#include<stdio.h>
int main( )
{
    int x, y;
    scanf("%d", &x);
    y=(x >5 ? 3 :4);
    printf("%d", y);
    return 0;
}
```

```
#include<stdio.h>
int main( )
{
    int x, y;
    scanf("%d", &x);
    if(x >5)
        y=3;
    else
        y=4;
    printf("%d", y);
    return 0;
}
```

/* Program for Addition (or) Multiplication*/

```
#include<stdio.h>
int main( )
{
        int   a, b, result, choice;
        printf("Enter  first number \n");
        scanf("%d",&a);
        printf("Enter second number\n");
        scanf("%d",&b);
        printf("Enter 1  for addition or 2 formultiplication\n");
        scanf("%d",&choice);
        result =(choice==1)?a+b:(choice==2)?a*b:printf("Invalid");
        if(choice==1||choice==2)
        printf("The result is %d\n\n",result);
        return 0;
}
```

Output

```
Enter first number
10
Enter second number
3
Enter 1 for addition or 2 for multiplication
2
The result is 30
```

/* Program to find the maximum of 3 Numbers*/

```
#include <stdio.h>
int main( )
{
    int a, b, c, max;
    printf("Enter three numbers: ");
    scanf("%d%d%d",&a, &b, &c);
    max =(a >b && a >c) ? a : (b >c) ? b :c;
    printf("\n Maximum between %d, %d and %d =%d", a, b, c,max);
    return 0;
}
```

Output

Enter three numbers: 30 10 40

Maximum between a, b and c = 40

2. 2 Operator Precedence

- ***Operator Precedence*** is used to determine the order of operators evaluated in an expression
  - Every operator has precedence (Priority)
  - Operator with higher precedence is evaluated first and the operator with least precedence is evaluated last
- ***Associativity*** is used when two operators of same precedence appear in an expression
  - Determines the order of evaluation of those operators
  - Associativity can be either Left to Right or Right to Left

2. 2 Operator Precedence Contd…

- Operators are listed in descending order of precedence
- An Expression can contain several operators with equal precedence
  - Evaluation proceeds according to the associativity of the operator i.e.,
    - From Right to Left (or)
    - From Left to Right
- ***Note:*** Order of operations is not defined by the language

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Precedence order	Operator	Associativity
1	( ) [ ] →	Left to right
2	++ -- - (unary) ! ~ * & sizeof	Right to left
3	* / %	Left to right
4	+ -	Left to right
5	<< >>	Left to right
6	< <= > >=	Left to right
7	== !=	Left to right
8	& (bitwise AND)	Left to right
9	^ (bitwise XOR)	Left to right
10	| (bitwise OR)	Left to right

2. 3 Expressions using Pre/Post Increment Operator

- Increment operators increase the value of the variable by one
- Decrement operators decrease the value of the variable by one
- ***Syntax***

  Increment operator: ++var_name; (or) var_name++;

  Decrement operator: –-var_name; (or) var_name –-;

- Example

  Increment operator : ++i ; i ++ ;

  Decrement operator : – – i ; i ––;

/* Expresssions using Pre-Increment Operator*/

```
#include<stdio.h>
int main( )
{
        int x,i;
        i=10;
        x=++i;
        printf("x: %d",x);
        printf("i: %d",i);
        return 0;
}
```

Output

x: 11

i: 11

/* Expresssions using Post-Increment Operator*/

```
#include<stdio.h>
int main( )
{
        int x,i;
        i=10;
        x=i++;
        printf("x: %d",x);
        printf("i: %d",i);
        return 0;
}
```

Output

x: 10

i: 11

/* Expresssions using Pre-Decrement Operator*/

```
#include<stdio.h>
int main( )
{
        int x,i;
        i=10;
        x=--i;
        printf("x: %d",x);
        printf("i: %d",i);
        return 0;
}
```

Output

```
x: 9
i: 9
```

/* Expresssions using Post-Decrement Operator*/

```
#include<stdio.h>
int main( )
{
        int x,i;
        i=10;
        x=i--;
        printf("x: %d",x);
        printf("i: %d",i);
        return 0;
}
```

Output

x: 10

i: 9

/* Expresssions using Increment / Decrement Operators*/

```
#include<stdio.h>
int main( )
{
        int  p,q,x,y;
        printf("Enter the value of x \n");
        scanf("%d" ,&x);
        printf("Enter the value of y \n");
        scanf("%d" ,&y);
        printf("x=%d\ny=%d\n",x,y);
        p=x++;
        q=y++;
        printf("x=%d\ty=%d\n",x,y);
        printf("x=%d\tq=%d\n",p,q);
        p=--x;
        q=--y;
        printf("x=%d\ty=%d\n",x,y);
        printf("p=%d\tq=%d\n",p,q);
        return 0;
}
```

Output

Enter the value of x 10

Enter the value of y 20

x = 10

y = 20

x = 11 y = 21

p = 10 q = 20

x = 10 y = 20

p = 10 q = 20

2. 4 Expressions using Conditional Operator

- Any operator is used on three operands or variable is known as **Ternary Operator**
- It can be represented with **?:**. It is also called as **conditional operator**

/* Program for Printing Odd or Even Number*/

```
#include<stdio.h>
int main( )
{
int num;
printf("Enter the Number : ");
scanf("%d",&num);
(num%2==0)?printf("Even\n"):printf("Odd");
}
```

Output

Enter the Number : 10

Even

/* Program for Eligibility to Vote*/

```
#include<stdio.h>
 int main( )
{
  int age;
   printf(" Please Enter your age here: \n ");
  scanf(" %d ", &age);
 (age >=18) ? printf(" You are eligible to Vote ") : printf(" You are not
                                                          eligible to Vote ");
 return 0;
}
```

Output

Please Enter your age here: 19

You are eligible to Vote

/* Program for Finding Biggest of 2 Numbers*/

```
#include<stdio.h>
int main( )
{
int a, b, max;
printf("Enter a and b: ");
scanf("%d%d", &a, &b);
max = a > b ? a : b; printf("Largest of the two numbers = %d\n", max);
return 0;
}
```

Output

Enter a and b: 10 20

Largest of the two numbers = 20

2. 5 Expressions using Assignment Operator

- Assignment Operator is used to assign value to an variable
- Assignment Operator is denoted by equal to sign
- Assignment Operator is binary operator which operates on two operands
- Assignment Operator have Two Values –L-Value and R-Value
  - Operator = copies R-Value into L-Value
- Assignment Operator have lower precedence than all available operators but has higher precedence than comma Operator

Operator	Example	Equivalent Expression
$=$	$m = 10$	$m = 10$
$+=$	$m += 10$	$m = m + 10$
$-=$	$m -= 10$	$m = m - 10$
$*=$	$m *= 10$	$m = m * 10$
$/=$	$m /=$	$m = m/10$
$\%=$	$m \% = 10$	$m = m\%10$
$<<=$	$a <<= b$	$a = a << b$
$>>=$	$a >>= b$	$a = a >> b$
$>>>=$	$a >>>= b$	$a = a >>> b$
$\&=$	$a \& = b$	$a = a \& b$
$\wedge=$	$a \wedge = b$	$a = a \wedge b$
$\mid=$	$a \mid = b$	$a = a \mid b$

2. 6 L-Value and R-Value of Expression

a) L-Value stands for **left value**

- L-Value of Expressions refer to a memory locations
- In any assignment statement L-Value of Expression must be a container(i.e. must have ability to hold the data)
- Variable is the only container in C programming thus L Value must be any Variable.
- L Value cannot be a Constant, Function or any of the available data type in C

2. 6 L-Value and R-Value of Expression Contd…

- Diagram Showing L-Value of Expression :

2. 6 L-Value and R-Value of Expression Contd…

```
#include<stdio.h>
int main( )
{
    int num;
    num=5;
    return(0);
}
```

Example of L-Value Expression

```
#include<stdio.h>
int main( )
{
    int num;
    5 =num; //Error
    return(0);
}
```

L-value cannot be a Constant

```
#include<stdio.h>
int main( )
{
    const num;
    num=20; //Error
    return(0);
}
```

L-value cannot be a Constant Variable

2. 6 L-Value and R-Value of Expression Contd…

```
#include<stdio.h>
#define MAX 20
int main( )
{
    MAX=20; //Error
    return(0);
}
```

L-value cannot be a MACRO

```
#include<stdio.h>
enum {JAN,FEB,MARCH};
int main( )
{
    JAN=20; //Error
    return(0);
}
```

L-value cannot be a Enum Constant

2. 6 L-Value and R-Value of Expression Contd…

b) RValue stands for **Right value** of the expression

- In any **Assignment statement** R-Value of Expression must be anything which is capable of returning Constant Expression or Constant Value

```
ivar  =   25 ;
```

L Value | Assignment Operator | R Value

2. 6 L-Value and R-Value of Expression Contd…

Examples of R-Value of Expression	
Variable	Constant
Function	Macro
Enum Constant	Any other data type

- Rvalue may be a Constant or Constant Expression
- Rvalue may be a MACRO
- RValue may be a variable

2.7 Control Statements

- Also called as Conditional Statement
- Decides order of execution based on conditions
- Helps repeat a group of statements
- Modifies control flow of program
- Decision Making
- Branching

2.7 Control Statements Contd...

- ***Types of Branching Statements***

a) if statement

   i. Simple if

   ii. if…else statement

   iii. nested if…else statement

   iv. else…if statement

b) switch statement

c) goto statement

2. 7 Control Statements Contd...

a) if statement

- Condition “True" - Statement block will be executed
- Condition “False“ - Statement block will not be executed.
- **Variations**

  i. Simple if

  ii. if…else statement

  iii. nested if…else statement

  iv. else…if statement

2.7 Control Statements Contd...

i. Simple if statement

- Basic if statement
- What is a condition?
- Executes statement block only if condition is true
- **Syntax**

```
if (condition)
{
        Statements;
}
```

```
if( condition)
 {
  statements;
 }
```

```
/* Simple if –Program to check whether a number is Odd*/
#include<stdio.h>
int main( )
{
    int number;
    printf("Enter the Number: ");
    scanf("%d, &number);
    if(number%2==0)
    {
        printf("The Number is Even");
    }
    return 0;
}
```

Output

Enter a value : 10342

The number is Even

2. 7 Control Statements Contd...

- **Try it Out Yourself ! Write a Cprogram to:**

1) Check whether the given number is Even
2) To check whether the given number is Greater
3) To check whether the given number is Smaller
4) To check whether the given number is positive
5) To check whether the given number is negative
6) To check whether the given number is zero
7) To check whether two numbers are equal

2. 7 Control Statements Contd...

ii. If else statement

- Extension of basic if statement
- Takes care of True and False condition
- Number of Statement Blocks - 2
  - Block 1 –True Condition
  - Block 2 –False Condition

```
if (condition)
{
        Statements;
}
Else
{
        Statements;
}
```

```
if( condition)
 {
  statements;
 }
else
 {
  statements;
 }
```

/* if else –Tocheck whether a number is Odd or Even*/

```
#include<stdio.h>
int main( )
{
    int number;
    printf("Enter the Number: ");
    scanf("%d, &number);
    if(number%2==0)
    {
        printf("The Number is Even");
    }
    else
    {
        printf("The Number is Odd");
    }
    return 0;
}
```

Output 1

Enter the Number : 10341

The number is Odd

Output 2

Enter the Number : 10342

The number is Even

2. 7 Control Statements Contd...

- **Try it Out Yourself ! Write a Cprogram to:**

1) To check whether the given number is Greater or Smaller
2) To check whether the given number is +ve or -ve
3) To check whether two numbers are equal or not

2.7 Control Statements Contd...

iii. Nested if else statement

- Used when a series of decisions are involved
- Makes a choice between several alternatives
- New if else statement block is used within existing if else statement block

```
if (test condition - 1)
  {
    if (test condition - 2)
      {
        statement 1;
      }
    else
      {
        statement 2;
      }
  }
else
  {
    statement 3;
  }
statement x;
```

True
condition 1
False
Statement 1
True
condition 2
False
Statement 2
True
condition 3
False
Statement 3
True
condition n
False
Statement n
Default statement
Statement x
Next statement

/*Program for Nested if else */

```
#include <stdio.h>

void main( )

{

    char username;

    int password;

    printf("Username:");

    scanf("%c",&username);

    printf("Password:");

    scanf("%d",&password);
```

```
if(username=='a')
        {
                if(password==12345)
                {
                        printf("Login successful");
                }
                else
                {
                printf("Password is incorrect, Try again.");
                }
        }
else
        {
        printf("Username is incorrect, Try again.");
        }
return 0;
}
```

Output 1

Username: a

Password: 12345

Login Successful

Output 2

Username: a

Password: 54321

Password is incorrect, Try again.

Output 3

Username: b

Password: 54321

Username is incorrect, Try again.

2. 7 Control Statements Contd...

- **Step 1:** First if condition will be true, if the user has typed 'a' as a username then the program control moves to second if condition and checks for the password
  - if it true it will print 'login successful'
  - else it will execute block statement 'Password is Incorrect, Try again.'.
- **Step 2:** If the first if condition is false then it executes last else block thus printing 'Username is Incorrect, Try again.'

2. 7 Control Statements Contd...

- **Step 3:** In this above example we have use username as single character to use multiple character username we need to use string data type

2. 7 Control Statements Contd...

b) Switch statement

- Allows to make decisions from a number of choices
- Also called as Switch-Case-Default Statement
- Faster than nested if else statement
- Easier to understand
- **Rules for writing switch ( ) statement**
  - Expression in switch must be an integer value or a character constant
  - No real numbers used in Expression

2.7 Control Statements Contd...

- Each case block and default block must end with break statements
- Default is optional
- Case keyword must end with colon (:)
- Default may be placed anywhere in the switch
- No two case constants are identical

2. 7 Control Statements Contd...

```
switch(variable or expression)
{
    case constant 1:
            statements;
    break;
    ....
    case constant N;
            statements;
    break;
    default:
            statements;
}
```

/* Program for Switch Case*/

```
#include<stdio.h>
int main( )
{
    int a, b, choice;
    printf("\nEnter Two Numbers: ");
    scanf("%d%d", &a,&b);
    printf("\n Enter 1  for Addition");
    printf("\n Enter 2 for Subtraction");
    printf("\n Enter 3 for Multiplication");
    printf("\n Enter 4 for Division");
    printf(" Enter your Choice");
    scanf("%d",&choice);
```

```
switch (choice)
{
    case 1:
        printf("Sum is : %d", a+b);
        break;
    case 2:
        printf("Difference is : %d", a-b);
        break;
    case 3:
        printf("Multiplication is : %d", a*b);
        break;
    case 4:
        printf("Difference is : %d", a/b);
        break;
```

```
            default:
                        printf("Invalid Choice:");
        }
getch( );
}
```

```
Enter two numbers
20
10
Enter 1  for Addition
Enter 2 for Subtraction
Enter 3 for Multiplication
Enter 4 for Division
Enter your Choice: 3
Product is : 200
```

2. 7 Control Statements Contd...

- **Nested Switch statement**
  - Inner switch( ) can be a part of an outer switch( )
  - Inner switch( ) and outer switch( ) case constants may be the same

/* Program for Nested Switch Case*/

```
#include<stdio.h>
int main( )
{
    int square, i, n, fact =1,choice;
    printf("\n Enter Any Number: ");
    scanf("%d", &n);
    printf(" 1. Square \n");
    printf(" 2. Factorial \n");
    printf(" 3. Find Odd or Even \n");
    printf(" 4. Exit \n");
    printf(" Enter your Choice");
    scanf("%d", &choice);
```

```
switch (choice)
{
    case 1:
        square = n * n;
        printf("The Square of the Given number is %d\n",
    square);
        break;
    case 2:
        for(i=1;i<=n;i++)
        {
                fact = fact * i;
        }
        printf("The Factorial of a given number is %d\n", fact);
        break;
```

```
        case 3:
        switch (n%2)
            {
                case 0:
                printf("Given Number is Even\n");
                case 1:
                printf("Given Number is Odd\n");
            }
            case   4:
            exit(0);
            default:
            printf("Invalid Choice. Please try again\n");
        }
    return 0;
}
```

```
Enter any number
5
1. Square
2. Factorial
3. Find Odd or Even
4. Exit
Enter your choice
2
The factorial of a given number is: 120
```

2. 7 Control Statements Contd...

c) The goto statement

- Transfers control from one point to another
- **Syntax**

```
goto label;
        statements;
        ..............
label
        statements;
```

2. 8 Looping Statements

- Loop –A segment of the program that is executed repeatedly until a condition is satisfied
- Classification –Entry Controlled & Exit Controlled
- **Types**
    a) while do loop
    b) do while loop
    c) for loop
        i. Nested for loop

2. 8 Looping Statements Contd...

a) The While Loop

- Simplest looping structure in C
- Statements in the program may need to repeat for many times. e.g., calculate the value of *n*!
- Loop consists of two segments
  - Control Statement
  - Body of the Loop
- How while loop works?

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

```
Initialize loop counter variable;
while (condition)
{
        Statements;
        increment /  Decrement loop
                     counter variable;
}
```

/* Program to Add 3 Numbers*/

```
#include<stdio.h>
int main( )
{
    int a, b, c, sum;
    printf("\n Enter the Three Numbers: ");
    scanf("%d%d%d", &a,&b,&c);
    sum = a+b+c;
    printf("The sum of 3 Numbers is %d", sum);
    return 0;
}
```

Output

Enter the Three Numbers: 10 20 30

The sum of 3 Numbers is: 60

/* Program to Add n Numbers*/

```
#include<stdio.h>
int main( )
{
    int i=1,n, sum=0;
    printf("\n Enter the value for n: ");
    scanf("%d", &n);
    while (i<=n)
    {
        sum =sum +i;
        i++;
    }
    printf("The sum of n Numbers is: %d", sum);
    return 0;
}
```

Output

Enter the value for n: 5

The sum of n Numbers is: 15

2. 8 Looping Statements Contd...

- **Try it Out Yourself! Write a C program to:**

1) To print all even numbers from 1 to 100
2) To print all even numbers from 1 to n
3) To print table for any number
4) To calculate the sum of its digits
5) To check whether the entered number is Prime or not
6) To get a number as input and print it in reverse.
7) To check whether the number is Armstrong number

2. 8 Looping Statements Contd...

b) The Do While Loop

- The body of the loop is executed at least once
- **Syntax**

```
do
{
        statements;
}
while (condition);
```

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

```
Initialize loop counter variable;
do
{
        Statements;
        increment /  Decrement loop
                counter variable;
}
while (condition)
```

2. 8 Looping Statements Contd...

While Do Loop	Do While Loop
Entry Controlled Loop	Exit Controlled Loop
Test condition is checked before body of the loop is executed	Test condition is checked after the body of the loop is executed
Loop will not be executed if condition is false	Loop will be executed at least once even if condition is false
Top tested loop	Bottom tested loop

2. 8 Looping Statements Contd...

while flowchart

do/while flowchart

2. 8 Looping Statements Contd...

c) The for loop

- Most commonly and popularly used loop structure
- Structure of the for loop
  - Initialize loop counter variable
  - Check for condition
  - Increment / Decrement the loop counter variable
- **Syntax**

for(initialization; condition; increment / decrement)

2. 8 Looping Statements Contd...

```
for( initialization; condition; incrementation )
        body;
```

2. 8 Looping Statements Contd...

- **Examples**

i.
```
for(i = 0; i < n; i++)
        {
                Statements;
        }
```

ii.
```
for(count = 0; count > n; count--)
        {
                Statements;
        }
```

/* Program to Add n Numbers using for loop */

```
#include<stdio.h>
int main( )
{
    int i, n, sum=0;
    printf("\n Enter the value for n: ");
    scanf("%d", &n);
    for (i =1; i<=n; i++)
    {
        sum =sum +i;
    }
    printf("The sum of n Numbers is: %d", sum);
    return 0;
}
```

Output

Enter the value for n: 5

The sum of n Numbers is: 15

2. 8 Looping Statements Contd...

- **Try it Out Yourself! Write a C program to:**

1) To print all even numbers from 1 to 100
2) To print all even numbers from 1 to n
3) To print table for any number
4) To calculate the sum of its digits
5) To check whether the entered number is Prime or not
6) To get a number as input and print it in reverse.
7) To check whether the number is Armstrong number

2. 9 Arrays

❑ Definition

An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

- ✓ Array is used to store a collection of data
- ✓ Array is a collection of variables of the same type.

First Element **Last Element**

Numbers[0]	**Numbers[1]**	**…….**	**Numbers[n]**

2. 9 Arrays Contd...

- Need for Arrays
  - Used to represent a list of numbers / names
  - Used to represent tabular data in 2, 3 or more dimensions
  - Important Data Structure in any programming language
- **Definition**
  - Collection of elements of similar data types
  - Each element is located in separate memory locations
  - Each Array element share a common name

2. 9 Arrays Contd...

❑ Characteristics of Arrays

- ❑All elements in the arrays share a common name
- ❑Elements distinguished by index number
- ❑Index (or) element number of an array plays vital role for calling each element
- ❑Specific array elements can be modified
- ❑Value of array element can be assigned to variables
- ❑Array elements stored in continuous memory locations

2. 9 Arrays Contd...

- Storage space for array depends on its data type and size

Total bytes = sizeof (Data type) x Size of Array

- Example

int a [5];

Total bytes = sizeof (int) x 5 = 2 x 5 = 10 bytes

2. 9 Arrays Contd...

a) Array Declaration

- Syntax

Datatype arrayname [size/subscript];

- **Data Type:** int, float, double, char, structure, union
- **Array Name:** Name given to the Array variable
- **Size / Subscript:** Number of values an Array can hold
- **Examples**

```
int numbers[5];        float marks[50];
char name[20];         double a[i];
```

2. 9 Arrays Contd...

- Illustration

int a[n];

2. 9 Arrays Contd...

- ***Static Array:*** Array size (range) declared in the program
- ***Dynamic Array:*** Array size given during execution

STATIC ARRAYS	DYNAMIC ARRAYS
Range / Size of an array included in the Array definition	Range / Size of an array not included in the Array definition
Static Arrays cannot be changed	Dynamic Arrays can be changed

2. 9 Arrays Contd...

b) Array Initialization

- **Initialization:** Assigning values to array elements
  - Values specified in curly braces separated by commas
- Examples

```
int a[ 5] ={1, 2, 3, 4, 5};
float b[3] ={ 40.5, 59.0, 98.5};
char name[6] =" SRMIST";
```

- Array element index start from 0

2. 9 Arrays Contd...

- Array elements are called by array names followed by the element numbers
- int a[5] ={1, 2, 3, 4, 5};

  a[0] refers to 1st element i.e., 1

  a[1] refers to 2nd element i.e., 2

  a[2] refers to 3rd element i.e., 3

  a[3] refers to 4th element i.e., 4

  a[4] refers to 5th element i.e., 5

2. 9 Arrays Contd...

c) Getting Input for Arrays

- Use for loops to get input in arrays
- Use for loops with regard to the Array's dimension
  - Input for One Dimensional Arrays –1 for loop

```
for(i = 0; i < 5; i++)
{
        scanf("%d", &a[i]);
}
```

2. 9 Arrays Contd...

- Input for Two Dimensional Arrays –2 for loops

```
for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
```

2. 9 Arrays Contd...

d) Printing Output in Arrays

- Use for loops to print array output
- Use for loops with regard to the Array's dimension
  - Printing One Dimensional Array Output –1 for loop for(i=0;i<5;i++)

```
{
        printf("%d",a[i]);
}
```

2. 9 Arrays Contd...

- Printing Two Dimensional Array Output –2 for loops

```
for(i =0; i <5; i++)
    {
        for(j=0; j <5; j++)
        {
            printff("%d", a[i][j]);
        }
    }
```

/* Program 1 : Array Declaration & Initialization*/

```
#include<stdio.h>
int main( )
{
    int i, arr[5];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;
    for(i=0; i<=n; i++)
    {
        printf("%d"\n, a[i]);
    }
    return 0;
}
```

Output

```
10
20
30
40
50
```

/* Program 2 : Array Declaration & Initialization*/

```
#include<stdio.h>
int main( )
{
    int i, arr[5];
    arr[5] ={10, 20, 30, 40, 50};
    for(i=0; i<=n; i++)
    {
        printf("%d", a[i]);
    }
    return 0;
}
```

Output

```
10
20
30
40
50
```

/* Program 3 : Array Declaration & Initialization*/

```
#include<stdio.h>
int main( )
{
    int i, n, arr[5];
    scanf("%d", &n);
    printf("Enter the Elements of Array\n");
     for(i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("The Elements of the Array are"\n");
```

```
    for(i=0; i<n; i++)
    {
        printf("%d", a[i]);
    }
    return 0;
}
```

Output

Enter the Elements of the Array

10 20 30 40 50

The Elements of the Array are

10 20 30 40 50

2. 9 Arrays Contd...

e) Classification of Arrays

i. One-Dimensional Array

ii. Two-Dimensional Array

iii. Multi-Dimensional Array

2. 9 Arrays Contd...

i. One Dimensional Array

- Data stored under a single variable using one subscript
- 1-D Array Declaration –Syntax

  datatype arrayname[size/subscript];

  - **Example:** int a [5];
- 1-D Array initialization –Syntax

  datatype arrayname[size] ={list of values};

  Example: int a [5] ={ 10, 20, 30, 40, 50};

/* Program 1 : One Dimensional Array*/

```
#include<stdio.h>
int main ( )
{
int a[10], n, i, sum;
clrscr( );
printf("Enter the Number of Elements\n");
scanf("%d", &n);
for(i = 0; i < n; i++)
{
      scanf("%d", & a [i]);
}
sum = 0;
for(i = 0; i < n;i++)
```

a [10]

a [0]	40
a [1]	22
a [2]	34
a [3]	12
a [4]	64
a [5]	
a [6]	
a [7]	
a [8]	
a [9]	

n	i
5	0

sum
0

/* Program 1 : One Dimensional Array*/

```
        {
            sum = sum + a[i];
        }
    printf("The Sum is: %d", sum);
    return 0;
    }
```

Output

Enter the Number of Elements

5

40 22 34 12 64

The Sum is 182

a [10]

a [0]	40
a [1]	22
a [2]	34
a [3]	12
a [4]	64
a [5]	
a [6]	
a [7]	
a [8]	
a [9]	

n	i
5	4

sum
182

/* Program 2 : 1-D Array for Sorting*/

```
#include<stdio.h>
int main( )
{
    int i, j, temp, n, a[10];
    printf("Enter the Number of Elements:");
    scanf("%d", &n);
    printf("Enter the Elements to be Sorted\n");
    for(i=0; i<n; i++)
    {
        scanf("%d\n", &a[i]);
    }
    for(i=0; i<n; i++)
    {
        for(j=i+1; j<n; j++)
        {
```

```
                    if(a[i] >a[j])
                    {
                              temp =a[i];
                              a[i] =a[j];
                              a[j] =temp;
                    }
          }
}
print("The Sorted Elements are: \n");
for(i=0; i<n; i++)
     {
          printf("%d\n", a[i]);
     }
return 0;
}
```

Output

Enter the Number of Elements:5

Enter the Elements to be Sorted

25

12

45

68

7

The Sorted Elements are:

7

12

25

45

68

THANK YOU

Unit-2

C Language :

C is a programing language developed at AT&T's Bell Laboratories of USA in 1972. It was designed & written by Dennis Ritchie.

C language is simple, more reliable, and easy to use.

In industry C language is still survived from last three decades.

Major part of popular ~~operation~~ operating system like windows, UNIX, Linux are still written in C.

The Character Set

Character denotes any alphabet, digit or special symbol used to represent information.

eg: Alphabets - A, B, - - - - - Y, Z
a, b, - - - - y, z

Digits - 0, 1, 2, 3 - - - - - 9

Special Symbol - ! @ # % & ( ) - _ { }
[ ] : ; " ' ? $

Constants, Variable & Keywords

→ A constant is an entity that doesn't change during the entire source code.

Rules for Integer Constants

(a) An integer constant must have at least one digit.

(b) It must not have a decimal point

(c) It can be either positive or negative.

(d) If no sign preceds an integer constant, it is assumed to be positive

No commas or blanks are allowed within integer constant.

(f) The allowable range for integer constant is -2147483648 to +2147483647. (for Visual Studio & gcc compiler)

for Turbo C (-32768 to 32767).

eg 426, +782, -8023

Real Constants:

Real constants are often called as floating point constants. Real constant could be written in two forms - Fractional & Exponential.

Rules

(a) A real constant must have at least one digit

(b) It must have a decimal point.

(c) It could be either positive or negative.

(d) Default sign is positive

(e) No comma or blank are allowed within a real constant.

eg: 325.34
426.0
-32.79

In exponential form of representation, the real constant is represented in two parts. Part appearing before 'e' is called mantissa whereas the part following 'e' is called exponent

Thus 0.000342 can be represented as 3.4e-4.

<u>Rules</u>:

(a) The mantissa part & the exponential part should be seprated by letter e or E.

(b) The mantissa part may have positive or negative sign.

(c) Default sign of mantissa is positive

(d) The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.

(e) Range of Real constants expressed in exponential form is -3.4e38 to 3.4e38

eg +3.2e-5
4.1e8
-0.2E+3

Character Constants

* A Character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
  Both the inverted commas should point left.

* Maximum length of a character constant is 1

  eg 'A', '5' '='

Variables:

An entity that may vary during program execution is called variable.
Variable names are given to location of memory, These locations can contain integer, real or character constants.

In any language, the type of variables that it can support depend on the type of constant that it can handle.

For example an integer variable can hold only integer constant, Real variable can hold Real constant & so on.

Rules for Constructing Variables Name

(a) A variable name is any combination of 1 to 31 alphabets, digits or underscore.

(b) The first ~~alphabet~~ character in variable name is always a alphabet or underscore.

(c) No commas or blanks are allowed within variable name.

(d) No special symbol other than underscore (_) can be used in variable name.

eg: int a, si, m_1;
float rate;
char code

Keywords

→ Keywords are the words whose meaning has already been explained to the C compiler

→ These keywords cannot be used as variable name

→ Some time these keywords are also known as reserved keywords or "Reserved words"

There are 32 keywords reserved in C-language.
ie

Auto	double	int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
Continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Data Type

Data Type indicate the type of data that a variable can hold. The data may be numeric, alphabatic, symbolic etc.

In C language the data type of any data is necessarly known in order to use them in program.

There are broadly two types of data types in C

→ Built in data types (Pre existing that are known to compiler)

→ User defined data types (These are defined by the

Built in Data Types

They are basic or primitive data type ~~of data that a variable can hold~~ and designate one single value. There are four basic fundamental built in data type in C

* Integer (int) size is 2 bytes
* Floating point (float) size is 4 bytes
* Double (double) size is 8 bytes
* Character (char) size is 1 byte.

Integer:

This is a keyword used to indicate an integer number which is sequence of digits without a decimal point.

The range of integer values lies between -32,768 to 32,767

eg -247, 14020, 27246, + 1966

Invalid

3,333 (comma not allowed)
-34.0 (decimal point)
+ 3,486.30 (comma & decimal)
999999 (out of range)

Float :

This is a keyword used to indicate the floating point number. Floating point number are same as decimal number but they include fractional part as well.

eg -263.238
2.63238E+02 $\{2.63238 \times 10^{2}\}$
2.63E-02 $\{2.63 \times 10^{-2}\}$

Char :

This is a keyword used to indicate the character type data. A character in C language is defined as the single character enclosed within a pair of apostrophes.

eg 'a', 'p', '2', '?'

Double :

This is a keyword used to indicate a double precision floating number. Normally it is equivalent to float, but the number of digits stored after decimal are 16 in double as compared to 6 in float.

eg 234.0000

Data Type Modifier

The basic data types can be modified using a series of type modifiers to fit the requirements of a particular program.

Signed

This can be applied to integer variables. The default declaration assumes a signed number. The range of signed integer lies between -32768 to 32767

Unsigned

It can be used for integer, It can be combined with long or short. If the keyword unsigned is used before integer the range of positive integers are doubled & negative integers are removed.

Range is 0 to 65535

Long: This is also applied to int data types when applied to int, it essentially doubles the length in bits of data type that it modifies.

It can also be used with double.

Long int ranges from -2147483648 to 2147483647

Short: This makes the size of an int half.

Default declaration of int is short.

Range is same as that of intgers in C language

Range is -32768 to +32767

Variable Declaration

⇒ If the variable is declared as <u>const variable</u>, its value cannot be changed. An attempt to modify its value will result in an error message.

eg: Const float PI = 3.142;

⇒ If the variable is defined as <u>volatile variable</u> or just as variable then its value can be modified at any time.

eg Volatile int x;

⇒ multiple assignment to the variables is possible in C language, It is possible to assign a value to multiple variables at same time (called as multiple assignment)

eg: int a = b = c = 20;
float x = y = z = 3.687;

Backslash Constant

→ A backslash constant is a combination of two characters in which the first character is always a backslash (\), and the second character can be any one of characters. a, b, n, t etc.

→ The backslash character constant are also called escape sequence.

→ The backslash constants are used in output statements.

eg

"\a"	System bell
"\b"	back space
"\n"	new line
"\t"	horizontal tab
"\v"	vertical tab
"\0"	null (end of string)

Symbolic Constants :

A symbolic Constant is a name that substitute a numeric constant, a character constant or string Constant.

eg

```
area = 3.142 * radius * radius;
```

here it can be defined as

```
# define PI 3.142
```

now. the formula is modified as

```
area = PI * radius * radius;
```

⇒ Generally, the symbolic constants are written in upper case letter.

⇒ Symbolic constant must be defined at the beginning of program. ie before main ( ).

Storage Class :

The storage class determines the part of the memory where the variable would be stored & how long the storage allocation continues to exist ie how long the variable would exist.

It also determines the scope of the variable which specifies the part of program where the variable name is visible.

Storage class also determines the initial value of the variable, if the variable is not explicitly initialized by any value.

① Automatic Storage Class

→ Variables declared at the start of a block are automatic variables

→ memory is allocated automatically upon entry to the block & freed automatically upon exiting the block

→ Variables are accessible only within the block, hence the scope of automatic variables are local to the block in which they are defined.

→ No block outside the defining block may have direct ~~acc~~ access to automatic variables.

→ By default all variables declared within a function are automatic

They can be declared as:

```
int i;
auto int i;
```

→ To declare any variable as automatic, a keyword auto is prefixed before its ~~name~~ declaration.

No Need

eg

```
# include <stdio.h>
void main ( )
{
    autoblock ( );
    autoblock ( );
}

auto block ( )
{
    auto int x = 10;
    printf ("\nvalue of x is %d", x);
    x = x+10;
}
```

Output is

Value of x is 10
Value of x is 10.

Note:

(i) Storage : Memory

(ii) Default initial value : Garbage value.

(iii) Scope : Local to block in which it is declared

(iv) Life : As long as program control remains within the block in which it is declared.

eg

```
# include <stdio.h>
void main ( )
{
auto int i, j ;
printf ("%d %d \n", i, j);
}
```

Output

1211 221 {garbage value}

eg

```
# include <stdio.h>
void main( )
{
    auto int i= 1;
    {
        auto int i=2;
        {
            auto int i=3;
            printf ("%d", i);
        }
        printf ("%d", i);
    }
    printf ("%d", i)
}
```

Output:

3 2 1

Register Storage Class

→ The register storage class specifier indicates to the compiler that the value of variable should reside in computer system register.

→ As the size of computer memory is limited, it is possible to keep only few variables, if compiler is not able to put variables in computer register, then the

variable are treated as automatic variables.

→ Thus it is not sure that the variable is stored in computer register.

→ Frequently used variables like loop counter are stored in registers.

Note :

Storage : CPU registers

Default Initial value: Garbage Value

Scope: local to block in which variable is declared

Life: Till control remains within the block in which the variable is defined

eg

```
# include <stdio.h>
void main ( )
  {
    register int i;
      for (i=1; i<=10; i++)
          printf ("%d", i);
  }
```

Static Storage Class

→ The static storage class provides a life time to the variables over the entire program, but it restricts the scope of static variable.

→ These variables are declared using the static keyword.

→ Default initial value of static variables is zero

→ Static variables retain their values even after the block in which they are defined terminates.

→ The value of a static variable in a function is retained between repeated call to the same function.

Note

Storage : Memory
Default initial value: Zero
Scope : Local to the block in which the variable is declared

Lifetime : Variables retains its value between different function calls.

eg

```
# include < stdio.h >
void main ( )
{
  increment ( );
  increment ( );
  increment ( );
}

increment ( )
{
  static int i = 1
  printf ("%d \n", i);
  i = i+1;
}
```

Output

In case of (auto int).	In case of (static int)
1	1
1	2
1	3

External Storage Class :

→ External storage class variables are accessible through out the program execution ie all the functions/ blocks use that particular variable.

→ These variables are known as global variables.

→ External variables are declared outside any functional block

→ Memory is allocated to these variables when program begins execution & the lifetime is untill the program terminates

→ Default initial value of these variables is zero.

→ Scope of these variables is global ie entire program code has accessibility to these variables

→ All functions may access external variable by its name, but if any variable declared within the function have same name, reference to the name access the local variables.

Note:

Storage: Memory
Default initial value: Zero
Scope: In all functions of program (global)
Lifetime: Throughout program execution

eg

```
# include < stdio.h>
int i;
void main()
  { printf ("\n i= %d", i);
    increment ( );
    increment ( );
    decrement ( );
    decrement ( );
  }
increment ( )
  { i= i+1;
  } printf ("\n on incremental i= %d", i);
decrement ( )
  { i= i-1;
  } printf ("\n on decrement i= %d", i);
```

output

```
i=0
on incremental i= 1
on incremental i=2
on decrement i= 1
```

Operators

→ In order to carry out basic arithmatic operations & logical operations operators are needed.

→ Operators act as connectors and they indicate what type of operation is being carried out.

→ The values that can be operated by these operator are called operands.

→ operators in C language are broadly classified in three categories.

Unary Operator

An operator that acts upon only one operand is known as unary operator. Some unary operators are:

① Unary minus

② Logical NOT operator (discussed in Logical operator)

③ Bitwise Complementation (discussed in Bitwise operator)

Unary Minus:

Any positive operand associated with unary minus operator gets its value changed to negative.

eg $a = 3, \quad b = 4$

$$c = a + (-b)$$
$$= 3 + (-4).$$
$$= -1$$

Binary Operator:

These operators acts upon two operands, hence named as binary. The binary operator are further divided in four categories

① Arithmatic Operator

② Logical Operator

③ Relational Operator

④ Bitwise operator

Arithmatic Operator:

Arithmatic operators performs basic arithmetic operations such as addition, substraction, multiplication & division.

Modulus operator is also used in C language that finds the remainder after integer division.

Operation	Operator	Precedence	Associativity
Addition	+	2	Left to Right
Substraction	−	2	Left to Right
multiplication	*	1	Left to Right
Division	/	1	Left to Right
Modulus	%	1	Left to Right

* The precedence indicates which terms to be evaluated.

* Associativity indicates the order in which the terms are evaluated.

Evaluation of Arithmatic Expression :

(i) a+b

⇒ There is only one operator (+), hence the value of 'a' is added to value of 'b'.

(ii) x*y/z

⇒ There are two operators ('*' & '/') having same priority. Now expression is scanned from left to right.

I^st x*y is evaluated

IInd The result of (x*y) is divided by z,

(iii) a+b*c/d-e

I^st b*c

IInd (b*c)/d

IIIrd a+((b*c)/d)

IVth (a+((b*c)/d))-e

eg:

```
int n1, n2;
n1 = 5;
n2 = 3;
```

(i) $n_1 + n_2 = 5 + 3 = 8$

(ii) $n_1 - n_2 = 5 - 3 = 2$

(iii) $n_1 * n_2 = 5 * 3 = 15$

(iv) $n_1 / n_2 = 5/3 = 1$

(v) $n_1 \% n_2 = 5 \% 3 = 2$

eg

```
float a, b;
a = 5.0
b = 2.0
```

(i) $a + b = 5.0 + 2.0 = 7.0$

(ii) $a - b = 5.0 - 2.0 = 3.0$

(iii) $a * b = 5.0 * 2.0 = 10.0$

(iv) $a / b = 5.0 / 2.0 = 2.5$

<u>Note:</u> % operator cannot be used with floating values

Rules for Evaluation of Arithmetic Expressions

① If the expression have paranthesses, then the expression inside parentheses must be evaluated first. (Rest priority & associativity will remain same)

② If unary minus is present in the expression then the term associated with unary minus must be evaluated before other expressions.

eg

$2 * ((i/3) + 4 * (j-2))$ where $i=8$ & $j=5$

Thus

$$2 * ((8/3) + 4 * (5-2)) = 2 * (2 + 4 \times 3)$$
$$= 2 * (2 + 12)$$
$$= 2 * 14$$
$$= \boxed{28}$$

Program to illustrates Basic Arithmetic Operators


```
# include <stdio.h>
# include <conio.h>
void main ( )
  {
    int a, b;
    int sum, dif, prod, quot, rem;
    printf (" Enter value of first number");
    scanf ("%d", &a);
    printf (" Enter value of second number");
    scanf ("%d", &b);

    sum= a+b;
    dif= a-b;
    prod = a*b;
    quot = a/b;
    rem = a%b;
    printf ("\n Sum = %d", sum);
    printf ("\n Difference=%d", dif);
    printf ("\n PRODUCT = %d", prod);
    printf ("\n Quotient = %d", quot);
    printf ("\n Remainder=%d", rem);
  }
```

Increment & Decrement Operator

* Increment operator is used to increment the value of an integer quantity by one.

→ This is represented by '++' symbol, This symbol can be placed before or after the integer variable.

ie

```
int a;
    a++ or ++a;   {both are allowed}
```

* Decrement operator is used to reduce the value of an integer quantity by one.

→ This is represented by '--' symbol. This symbol can be placed before or after integer variable.

ie

```
int b
    b-- or --b  {both are allowed}
```

Difference between Pre & Post increment/decrement

++a → indicates preincrement in the value of variable before it is used

a++ → indicates post increment in the value of variable after it is used

eg

```
# include <stdio.h>
void main( )
{
    int a=3, b=6;
    printf ("a = %d \n", a++);
    printf ("b = %d \n", ++b);
}
```

Output

a = 3
b = 7

Similarly

--b → decrement the value of variable first then use it

b-- → use the value of variable then decrement it

```
# include <stdio.h>
void main( )
{
  int m=6, n=6;
  printf("m=%d", --m);
  printf("\n n=%d", n--);
}
```

<u>output</u>

```
m = 5
n = 8
```

Relational operators

Relational operators are used to compare two operands, They also define the relationship existing between two constants or variables.

→ They result in either a TRUE or FALSE value. The value of TRUE is nonzero (ie 1) and the value of FALSE is zero

→ There are six relational operators in C language.

Operator	Meaning	Precedence	Associativity
<	is less than	1	Left to Right
<=	lesser than or equal to	1	Left to Right
>	greater than	1	Left to Right
>=	greater than or equal	1	Left to Right
==	is equal to	2	Left to Right
!=	not equal to	2	Left to Right

eg

```
# include <stdio.h>
void main ()
{
    int a = 8;
    int b = -5;
printf("%d < %d is %d\n", a, b, a<b);
printf("%d == %d is %d\n", a, b, a==b);
printf("%d != %d is %d\n", a, b, a!=b);
printf("%d > %d is %d\n", a, b, a>b);
printf("%d <= %d is %d\n", a, b, a<=b);
printf("%d >= %d is %d\n", a, b, a>=b);
}
```

Output

```
8 < -5 is 0
8 == -5 is 0
8 != -5 is 1
8 > -5 is 1
8 <= -5 is 0
8 >= -5 is 1
```

Logical Operator

Logical operators are used to take decisions, There are three such operators, (AND, OR & NOT).

→ AND, OR are binary operator & NOT is unary operator

→ The result of these operators is either TRUE (1) or FALSE (0).

→ Logical operator are used to connect one or more relational expression.

Operator	Meaning	Precedence	Associativity
&& (double ampersand)	Logical AND	2	Left to Right
|| (double pipeline)	Logical OR	3	Left to Right
! (exclamation)	Logical NOT	1	~~Left to Right~~ Right to Left

AND

Operand 1	Operand 2	operand 1 && operand2
0	0	0
0	1	0
1	0	0
1	1	1

OR

Operand 1	Operand2	operand 1 ~~&&~~ || Operand2
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Operand	! Operand
0	1
1	0

eg Suppose

$a=2, \quad b=4, \quad c=3$

```
a&&b||c && (!b)
2&&4||3 && (!4)
2&&4||3&&0
1||3&&0
1||0
=1
```

Prog

```
# include <stdio.h>
void main( )
{
    int a=4, b=3, c;
    c= a&&b;
    b= a||b||c;
    a= a&&b||c;
    printf ("%d.%d %d", a, b, c);
}
```

Output

1, 1, 1

Bitwise Operator :

All the data stored in the computer's memory as a sequence of bits (ie 0, 1), some application needs manipulation of these bits.

→ To perform bitwise operations, C provides six operators.

→ These operators works on char & int variables but not on floating type data.

~~There~~ .

Operator	Symbol Name	Meaning
&	Ampersand	Bitwise AND
|	Pipeline	Bitwise OR
^	Caret	Bitwise XOR
~	Tide	1's complement (NOT)
<<	Double less than	Left shift of bits
>>	Double greater than	Right shift of bits

XOR

operand 1	Operand2	operand1 ^ operand2
0	0	0
0	1	1
1	0	1
1	1	0

eg int a=4, b=3;

For 8 bit Computer

Binary Equivalent of 4 & 3 are

a= 0000 0100

b= 0000 0011

now

a & b = 0000 0000

a | b = 0000 0111

a ^ b = 00000111

Bitwise Complement :

Let a=10

then a = 1010

if b = ~a = ~(1010)

then b = 0101

Bitwise Left shifting

In left or right shift operator case, the general form of representation is

<variable>	<shift operator>	<nob>

variable → int or char data element
shift operator → left shift or right shift operator
nob → number of bits to shift

eg: (i) a >> 1 or (ii) a << 2

Left shift

Let a = 132
or 10000100

Before left shifting

1	0	0	0	0	1	0	0

After one bit shifted to left

0	0	0	0	1	0	0	0

Right Shift

Before shifting

1	0	0	0	0	1	0	0

After one bit Right shifting

0	1	0	0	0	0	1	0

Program

```
# include < stdio.h >
Void main ( )
  {
   int x, y;
   x=128, y= 32;
  x = x>>1;
printf ("After right shifting by 1, x= %d \n", x);
  y = y<<2;
printf ("After Left shifting by 2, y= %d \n", y);
  }
```

Output

After right shifting by 1, x = 64

After left shifting by 2, y = 128

Ternary Operator

* Ternary operator is conditional operator in C.
* It takes three operands.
* '?' is used as a ternary operator in C
* The general form of a ternary expression is as follows:

<expression> ? <value1> : <value2>

where,

expression → Relational expression
value1 → value to be assigned on true
value2 → value to be assigned when the result of expression is false

eg: c = a<b ? a : b;

here c is assigned a value of a if a is less than b. Otherwise it will be assigned the value of b.

Prog

```
# include < stdio.h>
void main ()
{
    int a=4, b=5, result1, result2;
    result1 = a>b ? a : b;
printf ("The result1 = %d \n", result1);
```

```
result2 = a<b? a:b;
    printf ("The result 2= %d\n", result2);
}
```

<u>Output</u>:

The result 1 = 5

The result 2 = 4

Special Operator of C

<u>Comma Operator</u>: The comma operator is basically associated with for statement. It is also used to link two or more related expressions together. In such case the expression are evaluated in left to right fashion and we can obtain the value of the combined expression ~~from the value of the combined expression~~ from the value of rightmost expression.

eg. Sum = (a=12, b=22, a+b).

here there are 3 expressions, the value of last expression is set to sum.

Size of ( ) operator

→ The size of() operator returns the size (in number of bytes) of the operand.

→ It is written in lower case letters.

→ It must precede its operand.

→ The operand may be constant, variable or data type.

eg

```
x = size of (int);
y = size of (sum);
```

Program

```
void main()
{ int x = 10;
  float y = 6.5;
  Char ch = 'y';
printf("size of x = %d\n", size of (x));
printf("size of y = %d\n", size of (y));
printf("size of ch = %d\n", size of (ch));
printf("size of double = %d\n", size of (double))
}
```

Output

size of x = 2

size of y = 4

size of ch = 1

size of double = 8

Precedence & Associativity of all C operators

Operator Category	Operator	Precedence	Associativity
Parentheses, braces	(), []	1	Left to Right
Unary operator	+, -, ++, --, !, ~, &	2	Right to Left
multiplicative operator	*, /, %	3	Left to Right
Additive Operators	+, -	4	Left to Right
Shift operator	<<, >>	5	Left to Right
Relational operator	<, <=, >, >=	6	Left to Right
Equality operators	==, !=	7	Left to Right
Bitwise operators	&, ^, |	8	Left to Right
Logical operators	&&, ||	9	Left to Right
Conditional operator	?, :	10	Right to left
Assignment operator	=, +=, -=, *=	11	Right to left
Comma operator	,	1	Left Right

Assignment Operator:

→ Assignment operator assigns the value of expression on the right side of it to the variable on the left of it.

→ The assignment expression evaluates the operand on the right side of operator (=) & places its value in the variable on the left.

eg

```
a=4;
area= length * breadth;
```

C language also provides a compact way of writing assignment statements in expressions.

The general form of shorthand assignment in C is as follows:

Var	operator	=	expression	;

Var → Variable

operator → an arithmetic or bitwise operator

expression → is simply an expression.

Operator	Assignment (a & b)	Shorthand assignment
+	a = a + b	a += b
−	a = a − b	a − = b
*	a = a * b	a * = b
/	a = a / b	a / = b
%	a = a % b	a % = b

Data Type Conversion :

Some time it is needed to change the datatype of variable. When we change the nature of data stored in variable. This process is known as data type conversion or type casting.

Syntax (datatype) variable

⇒ eg

```
int K= 2;
float term;
term = 1/K ;
```

Here the value assigned to the term is 'zero' as of integer division.

But the real value which should be 0.5
25

for 0.5 datatype conversion is required which can be achieved as follows:

```
term = 1/(float)K;
```

Mixed Mode Operations :

⇒ Expressions that involves, elements of type real & integer are called mixed mode expressions

→ If an expression containing both integer & float values, then the result will be of type float.

→ If the result value is assigned to integer then the fractional part of float must be truncated.

→ Operators that can be used in mixed mode expressions are (+, -, *, /, %)

eg:

int a = 4, b = 2;

float x = 2, y = 3

(i) a = (x/y) + (a/b);

(ii) b = (a*x)/y;

(iii) x = a - b;

Solⁿ

(i)

$$a = (x/y) + (a/b)$$
$$= (2.0/3.0) + (4/2)$$
$$= 0.6 + 2$$
$$= 2.6$$

$$\boxed{a = 2}$$

(ii)

$$b = (a*x)/y$$
$$4 \times 2.0$$
$$= 8.0/3$$
$$= 2.66$$

$$\boxed{b = 2}$$

(iii)

$$x = a - b$$
$$= 4 - 2$$
$$= 2$$

$$\boxed{x = 2.0}$$

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

18CSS101J – Programming for Problem Solving

Unit III

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

UNIT III

Initializing and accessing of 2D Array – Initializing multidimensional Array – Array programs 2D – Array contiguous memory – Array advantages and Limitations – Array construction for real-time application common programming errors – String Basics – String Declaration and Initialization – String Functions: gets(), puts(), getchar(), putchar(), printf() - String Functions: atoi, strlen, strcat strcmp – String Functions: sprintf, sscanf, strrev, strcpy, strstr, strtok – Arithmetic characters on strings.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

UNIT III

- Functions declaration and definition : Types: Call by Value, Call by Reference – Function with and without Arguments and no Return values – Functions with and without Arguments and Return Values – Passing Array to function with return type – Recursion Function

3.1INITIALIZING AND ACCESSING OF 2D ARRAY

3.1.1 ACCESSING OF 2D ARRAY

- Two Dimensional Array requires ***Two Subscript*** Variables
- Two Dimensional Array stores the values in the form of matrix.
- One Subscript Variable denotes the "***Row***" of a matrix.
- Another Subscript Variable denotes the "***Column***" of a matrix.

3.1.2 INITIALIZING OF 2D ARRAY

An array of two dimensions can be declared asfollows:

data_type array_name[size1][size2];

Here ***data_type*** is the name of some type of data, such as int. Also, ***size1*** and ***size2*** are sizes of the array's first and second dimensions respectively.

Declaration and Use of Two Dimensional Array :

int a[3][4];

Use :

```
for(i=0;i<row,i++)
   for(j=0;j<col,j++)
   {
   printf("%d",a[i][j]);
}
```

Meaning of Two Dimensional Array :

- Matrix is having 3 rows (i takes value from 0 to 2)
- Matrix is having 4 Columns (j takes value from 0 to 3)
- Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.
- Name of 2-D array is ‘a‘ and each block is identified by the row & column number.
- Row number and Column Number Starts from 0.
- If a[0][0] means 0th row 0th column, a[0][1] means 0th row 1st column, a[0][2] means 0th row 2nd column like wise it goes on…….

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Initializing 2D array:

- Row wise assignment
- Combine assignment
- Selective assignment

Method 1 : INITIALIZING ALL ELEMENTS ROWWISE

For initializing 2D Array we can need to assign values to each element of an array using the below syntax.

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

Consider the below program –

```
#include<stdio.h>
int main(){
  int i, j;
  int a[3][2] ={ {1, 4 },
                 { 5, 2},
                 { 6, 5}};
  for (i =0; i <3; i++) {
    for (j =0; j<2; j++) {
      printf("%d ", a[i][j]);
    }
    printf("\n");
  } return 0;
}
```

Output :

```
1 4
5 2
6 5
```

We have declared an array of size 3 X2, It contain overall 6 elements.

Row 1 : { 1 , 4},

Row 2 : { 5 , 2},

Row 3 : { 6 , 5}

We have initialized each row independently

a[0][0] = 1

A[0][1] = 4

Method 2 : COMBINE ASSIGNMENT

Initialize all Array elements but initialization is much straight forward. All values are assigned sequentially and row-wise

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

So here it automatically assigns that number 3 has row and number 2 has column.

Consider the below program –

```
#include <stdio.h>
int main(){
  int i, j;
  int a[3][2] ={ 1, 4, 5, 2, 6, 5 };
  for (i =0; i <3; i++) {
    for (j = 0; j < 2; j++) {
      printf("%d ", a[i][j]);
    }
    printf("\n");
  }  return 0;
}
```

Output :

1 4

5 2

6 5

Method 3: SELECTIVE ASSIGNMENT

```
int a[3][2] ={
{ 1 },
{ 5 , 2},
{ 6 }
};
```

Now we have again going with the way 1 but we are removing some of the elements from the array. In this case we have declared and initialized 2-D array like this

Consider the below program –

```
#include <stdio.h>
int main(){
  int i, j;
  int a[3][2] ={ { 1},
                 { 5, 2},
                 { 6}};
  for (i =0; i <3; i++) {
    for (j =0; j<2; j++) {
      printf("%d ", a[i][j]);
    } printf("\n"); }
  return 0;
}
```

Output :

1 0
5 2
6 0

3.2 INITIALIZING MULTIDIMENSIONAL ARRAY

Arrays with more than one dimension are called multidimensional arrays.

Declaration of three-Dimensional arrays:

data_type array_name[size1][size2][size3];

Arrays do not have a shaped like squares and cubes; each dimension of the array be given in different size as follows:

Eg: int non_cube[2][[6][8];

You can initialize a three dimensional array in a similar way like a two dimensional array. Here's an example,

```
int test[2][3][4] ={
        { {3, 4, 2, 3}, {0, -3, 9, 11}, {9, 1, 6, 2} },
        { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
        };
```

In this program we mention size1 – row, size2 – column, size3 – number of elements in that array.

CProgram to store values entered by the user in a three-dimensional array and display it.

```
#include <stdio.h>
int main()
{
  int i, j, k, test[2][3][2]; // this array can store 12 elements
   printf("Enter 12 values: \n");
   for(i = 0; i < 2; ++i) {
      for (j = 0; j < 3; ++j) {
         for(k = 0; k < 2; ++k ) {
            scanf("%d", &test[i][j][k]);
         }
```

```
}
}
printf("\nDisplaying values:\n");//Displaying values with
                                  proper  index.
   for(i = 0; i < 2; ++i){
      for (j = 0; j < 3; ++j) {
         for(k = 0; k < 2; ++k ) {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
                  }
         }
}
   return 0;
}
```

OUTPUT:

Enter 12 values:

1
2
3
4
5
6
7
8
9
10
11
12

Displaying Values:

test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12

3.3 ARRAYPROGRAMS – 2D

a) Basic 2d array program

b) Store and display values

c) Sum of two matrices using Two dimensional arrays

d) Transpose of Matrix

e) Multiplication of 2d matrices.

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

Basic 2d array program

```
#include<stdio.h>
int  main(){
    int disp[2][3]; /* 2D array
declaration*/
    int i, j; /*Counter variables
for loop*/
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("Enter value for
disp[%d][%d]:", i, j);
      scanf("%d", &disp[i][j]);
    } }
```

```
//Displaying array elements
  printf("Two Dimensional array
elements:\n");
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
        printf("%d ", disp[i][j]);
         if(j==2){
printf("\n");
      }
    } }
  return 0; }
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

Basic 2d array program

OUTPUT

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
        1 2 3
        4 5 6
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

STORE AND DISPLAY VALUES OF ARRAY

```
#include <stdio.h>
const int CITY = 1;
const int WEEK = 7;
int main()
{
   int temperature[CITY][WEEK];
   for (int i = 0; i < CITY; ++i){
      for(int j = 0; j < WEEK; ++j) {
         printf("City %d, Day %d: ",
i+1, j+1);
         scanf("%d",
&temperature[i][j]);
         }
         }
printf("\nDisplaying values:
\n\n");
   for (int i = 0; i < CITY; ++i) {
      for(int j = 0; j <WEEK; ++j)
      {
         printf("City %d, Day %d =
%d\n", i+1, j+1,
temperature[i][j]);
      }
   }
   return 0;
}
```

OUTPUT

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
```

```
Displaying values:
City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

SUM OF 2D ARRAY

```
#include <stdio.h>
int main()
{
  float a[2][2], b[2][2], c[2][2];
  int i, j;   // Taking input using nested for loop
  printf("Enter elements of 1st matrix\n");
  for(i=0; i<2; ++i)
   for(j=0; j<2; ++j)
   {
     printf("Enter a%d%d: ", i+1, j+1);
     scanf("%f", &a[i][j]);
   } // Taking input using nested for loop
  printf("Enter elements of 2nd matrix\n");
  for(i=0; i<2; ++i)
   for(j=0; j<2; ++j)
   {
     printf("Enter b%d%d: ", i+1, j+1);
     scanf("%f", &b[i][j]);
   }// adding corresponding elements of two arrays
  for(i=0; i<2; ++i)
   for(j=0; j<2; ++j)
   {
     c[i][j] = a[i][j] + b[i][j];
   }
```

```
//  Displaying the sum
   printf("\nSum Of Matrix:");
   for(i=0; i<2; ++i)
    for(j=0; j<2; ++j)
    {
      printf("%.1f\t", c[i][j]);
           if(j==1)
        printf("\n");
    }
return 0;
}
```

OUTPUT:

```
Enter elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;
Sum Of Matrix:
2.2     0.5
-0.9    25.0
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

TRANSPOSE OF MATRIX

```
#include <stdio.h>
int main()
{
   int a[10][10], transpose[10][10], r, c,
i, j;
   printf("Enter rows and columns of
matrix: ");
   scanf("%d %d", &r, &c); // Storing
elements
   printf("\nEnter elements of
matrix:\n");
   for(i=0; i<r; ++i)
      for(j=0; j<c; ++j)
      {
       printf("Enter element a%d%d:
",i+1, j+1);
         scanf("%d", &a[i][j]);      }
   printf("\nEntered Matrix: \n");
   for(i=0; i<r; ++i)
      for(j=0; j<c; ++j)
      {
         printf("%d ", a[i][j]);
         if (j ==c-1)
            printf("\n\n");
      }// Finding the transpose of
matrix a
   for(i=0; i<r; ++i)
      for(j=0; j<c; ++j)
      {
         transpose[j][i] =a[i][j];
      }
```

```
// Displaying the transpose of matrix a
   printf("\nTranspose of Matrix:\n");
   for(i=0; i<c; ++i)
      for(j=0; j<r; ++j)
      {
         printf("%d  ",transpose[i][j]);
         if(j==r-1)
            printf("\n\n");
      }
   return 0;
}
```

output

```
Enter rows and columns of matrix: 2  3
Enter element of matrix:
Enter element a11: 2
Enter element a12: 3
Enter element a13: 4
Enter element a21: 5
Enter element a22: 6
Enter element a23: 4
Entered Matrix:
2  3  4
5  6  4
Transpose of Matrix:
2  5
3  6
4  4
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

MULTIPLCATION OF 2D MATRIX

```
#include <stdio.h>
int main()
{
 int m, n, p, q, c, d, k, sum=0;
 int first[10][10], second[10][10], multiply[10][10];
  printf("Enter number of rows and columns of first matrix\n");
scanf("%d%d", &m, &n);
printf("Enter elements of first matrix\n");
  for (c=0; c<m; c++)
   for (d=0; d<n; d++)
     scanf("%d", &first[c][d]);
  printf("Enter number of rows and columns of second matrix\n");
scanf("%d%d", &p, &q);
 for (c=0; c<p; c++)
     for (d=0; d<q; d++)
       scanf("%d", &second[c][d]);
    for (c = 0; c < m; c++) {
     for (d = 0; d < q; d++) {
     for (k=0; k<p; k++){
       sum = sum + first[c][k]*second[k][d];
      }
       multiply[c][d] =sum;
      sum=0;
     } }
```

```
 printf("Product of the matrices:\n");
    for (c = 0; c < m; c++){
     for (d = 0; d < q; d++)
       printf("%d\t", multiply[c][d]);
      printf("\n");
   } }
  return 0;
}
```

OUTPUT

Enter the number of rows and columns of first matrix: 3 3
Enter the elements of first matrix: 1 2 0
0 1 1
2 0 1
Enter the number of rows and columns of second matrix:
3
3
Enter the elements of second matrix:
1 1 2
2 1 1
1 2 1
Product of entered matrices:-

5	3	4
3	3	2
3	4	5

4. ARRAY CONTIGUOUS MEMORY

- When Big Block of memory is reserved or allocated then that memory block is called as Contiguous Memory Block.
- Alternate meaning of Contiguous Memory is continuous memory.
- Suppose inside memory we have reserved 1000-1200 memory addresses for special purposes then we can say that these 200 blocks are going to reserve contiguous memory.

How to allocate contiguous memory:

- Using static array declaration.
- Using calloc() / malloc() function to allocate big chunk of memory dynamically.

Contiguous Memory Allocation

- Two registers are used while implementing the contiguous memory scheme. These registers are base register and limit register.

When OS is executing a process inside the main memory then content of each register are as–

- Register - Content of register.
- Base register- Starting address of the memory location where process execution is happening.
- Limit register- Total amount of memory in bytes consumed by process.

When process try to refer a part of the memory then it will firstly refer the base address from base register and then it will refer relative address of memory location with respect to base address

3.5 ADVANTAGE AND LIMITATIONS OF ARRAY

1. Advantages:

- It is better and convenient way of storing the data of same datatype with same size.
- It allows us to store known number of elements in it.
- It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc.
- It allows to store the elements in any dimensional array - supports multidimensional array.

3.5.2 Limitations of Array:

A) Static Data

- Array is Static data Structure
- Memory Allocated during Compile time.
- Once Memory is allocated at Compile Time it Cannot be Changed during Run-time.

B) Can hold data belonging to same Data types

- Elements belonging to different data types cannot be stored in array because array data structure can hold data belonging to same data type.
- Example : Character and Integer values can be stored inside separate array but cannot be stored in single array

C) Inserting data in Array is Difficult

- Inserting element is very difficult because before inserting element in an array have to create empty space by shifting other elements one position ahead.
- This operation is faster if the array size is smaller, but same operation will be more and more time consuming and non-efficient in case of array with large size.

D) Deletion Operation is difficult

- Deletion is not easy because the elements are stored in contiguous memory location.
- Like insertion operation , we have to delete element from the array and after deletion empty space will be created and thus we need to fill the space by moving elements up in the array.

E) Bound Checking

- If we specify the size of array as 'N' then we can access elements upto 'N-1' but in C if we try to access elements after 'N-1' i.e Nth element or N+1th element then we does not get any error message.
- Process of Checking the extreme limit of array is called *Bound checking* and Cdoes not perform Bound Checking.
- If the array range exceeds then we will get garbage value as result.

E) Shortage of Memory

- Array is Static data structure. Memory can be allocated at *compile time* only Thus if after executing program we need more space for storing additional information then we cannot allocate additional space at run time.
- Shortage of Memory , if we don't know the size of memory in advance

F) Wastage of Memory

- Wastage of Memory , if array of large size is defined.

3.6 ARRAY CONSTRUCTION FOR REAL-TIME APPLICATION COMMON PROGRAMMING ERRORS

(i) Constant Expression Require

```
#include<stdio.h>
void main()
{
int i=10;
int a[i];
}
```

In this example we see what's that error?

- We are going to declare an array whose size is equal to the value of variable.
- If we changed the value of variable then array size is going to change.
- According to array concept, we are allocating memory for array at compile time so if the size of array is going to vary then how it is possible to allocate memory to an array.
- *i is initialized to 10 and using a[i] does not mean a[10] because 'i' is Integer Variable whose value can be changed inside program.*

- **Value of Const Variable Cannot be changed**
- we know that value of Const Variable cannot be changed once initialized so we can write above example as below –

```
#include<stdio.h>
void main()
{
const int i=10;
int a[i];
}
```

or

```
int a[10];
```

(ii) Empty Valued 1D Array

```
#include<stdio.h>
void main()
{
int arr[];
}
```

Instead of it Write it as–

```
#include<stdio.h>
void main()
{
int a[] ={1,1};
}
```

- Consider this example, we can see the empty pair of square brackets means we haven't specified size of an 1D Array. In this example array 'arr' is undefined or empty.
- Size of 1D Array should be Specified as a Constant Value.

```
#include<stdio.h>
void main()
{
int a[] = {};// This also Cause an Error
}
```

(iii) 1D Array with no Bound Checking

```
#include<stdio.h>
void main()
{
int a[5];
printf("%d",a[7]);
}
```

Here Array size specified is 5.

- So we have Access to Following Array Elements – a[0],a[1],a[2],a[3] and a[4].
- But accessing a[5] causes Garbage Value to be used because C Does not performs Array Bound Check.

If the maximum size of array is “MAX” then we can access following elements of an array –

Elements accessible for Array Size "MAX" = arr[0]
=.
= arr[MAX-1]

4. Case Sensitive

```
#include<stdio.h>
void main()
{
int a[5];
printf("%d",A[2]);
}
```

Array Variable is Case Sensitive so A[2] does not print anything it Displays Error Message : “Undefined SymbolA“

3.7 STRING BASICS

- **Strings in Care represented by arrays of characters.**
- String is nothing but the collection of the individual array elements or characters stored at contiguous memory locations

  i) Character array – 'P','P','S'

  ii) Double quotes - “PPS" is a example of String.

  - If string contains the double quote as part of string then we can use escape character to keep double quote as a part of string.
  - “PP\"S" is a example of String

iii) Null Character

- The end of the string is marked with a special character, the *null character*, which is a character all of whose bits are zero i.e., a NULL.
- String always Terminated with NULL Character ('\0')

char name[10] ={'P','P','S','\0'}

  - NULL Character is having ASCII value 0
  - ASCII Value of '\0' =0
  - As String is nothing but an array , so it is Possible to Access Individual Character

name[10] = "PPS";

- It is possible to access individual character

name[0] = 'P';

name[1] = 'P';

name[2] = 'S';

name[3] = '\0';

iv) MEMORY

Each Character Occupy 1 byte of Memory

Size of "PPS" = Size of 'P' +

= Size of 'P' +

= Size of 'S' ;

Size of "PPS" is 3 BYTES

Each Character is stored in consecutive memory location.

Address of 'P' = 2000

Address of 'P' = 2001

Address of 'S' = 2002

8. STRING DECLARATION AND INITIALIZATION

- ***3.8.1 String Declaration:***
  - String data type is not supported in C Programming. String means Collection of Characters to form particular word. String is useful whenever we accept name of the person, Address of the person, some descriptive information. We cannot declare string using String Data Type, instead of we use array of type character to create String.
  - Character Array is Called as 'String'.
  - Character Array is Declared Before Using it in Program.

char String_Variable_name [SIZE] ;

Eg: char city[30];

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

Point	Explanation
Significance	- We have declared array of character[i.e String]
Size of string	- 30 Bytes
Bound checking	- CDoes not Support Bound Checking i.e if we store City with size greater than 30 then Cwill not give you any error
Data type	- char
Maximum size	- 30

Precautions to be taken while declaring Character Variable :

- String / Character Array Variable name should be legal CIdentifier.
- String Variable must have Size specified.
  - **char city[];**
- Above Statement will cause compile time error.
  - Do not use String as data type because String data type is included in later languages such as C++ / Java. C does not support String data type
  - **String city;**
- When you are using string for other purpose than accepting and printing data then you must include following header file in your code –
  - **#include<string.h>**

2. *Initializing String [Character Array] :*

- Whenever we declare a String then it will contain garbage values inside it. We have to initialize String or Character array before using it. Process of Assigning some legal default data to String is Called Initialization of String. There are different ways of initializing String in C Programming –

  1)Initializing Unsized Array of Character
  2)Initializing String Directly
  3)Initializing String Using Character Pointer

Way 1 : Unsized Array and Character

- ***Unsized Array*** : Array Length is not specified while initializing character array using this approach
- Array length is Automatically calculated by Compiler
- Individual Characters are written inside Single Quotes , Separated by comma to form a list of characters. Complete list is wrapped inside Pair of Curly braces
- ***NULL Character*** should be written in the list because it is ending or terminating character in the String/Character Array
- char name [] ={'P','P','S','\0'};

Way 2 : Directly initialize String Variable

- In this method we are directly assigning String to variable by writing text in double quotes.
- In this type of initialization , we don't need to put NULL or Ending / Terminating character at the end of string. It is appended automatically by the compiler.
- char name [ ] = "PPS";

Way 3 : Character Pointer Variable

- Declare Character variable of pointer type so that it can hold the base address of “String”
- Base address means address of first array element i.e (address of name[0])
- NULL Character is appended Automatically
- char *name = "PPS";

9. STRING FUNCTIONS

- gets()
- puts()
- Getchar()
- Putchar()
- Printf()
- Atoi()
- Strlen()
- Strcat ()
- Strcmp()
- Sprintf()
- Sscanf()
- Strrev()
- Strcpy()
- Strstr()
- strtok()

3.9.1 GETS():

Syntax for Accepting String :

```
char * gets ( char * str );
O
R gets( <variable-name> )
```

Example :

```
#include<stdio.h>
void main()
{
char name[20];
printf("\nEnter the Name :");
gets(name); }
```

Output:

Enter the name: programming in c

Explanation :

- Whenever gets() statement encounters then characters entered by user (the string with spaces) will be copied into the variable.
- If user start accepting characters , and if new line character appears then the newline character will not be copied into the string variable(i.e name).
- A terminating null character is automatically appended after the characters copied to string vriable (i.e name)
- gets() uses stdin (Standered Input Output) as source, but it does not include the ending newline character in the resulting string and does not allow to specify a maximum size for string variable (which can lead to buffer overflows).

Some Rules and Facts :

A. %sis not Required :

Like scanf statement %s is not necessary while accepting string.

```
scanf("%s",name);
```

and here is gets() syntax which is simpler than scanf() –

```
gets(name);
```

B. Spaces are allowed in gets() :

```
gets(name);
```

Whenever the above line encounters then interrupt will wait for user to enter some text on the screen. When user starts typing the characters then all characters will be copied to string and when user enters newline character then process of accepting string will be stopped.

Sample Input Accepted by Above Statement :

- Value Accepted : Problem solving\n
- Value Stored : Problem solving (\n Neglected)

2. *PUTS():*

Way 1 :Messaging

- puts(" Type your Message / Instruction ");
- Like Printf Statement puts() can be used to display message.

Way 2 : Display String

- puts(string_Variable_name) ;

Notes or Facts :

- puts is included in header file “stdio.h”
- As name suggest it used for Printing or Displaying Messages or Instructions.

Example :

```
#include< stdio.h>
#include< conio.h>
void main()
 {
  char string[] = "This is an example string\n";
  puts(string);     // String is variable Here
  puts("String");   // String is in Double Quotes
  getch();
}
```

Output :

String is : This is an example string

String is : String

3.9.3 GETCHAR():

- Getchar() function is also one of the function which is used to accept the single character from the user.
- The characters accepted by getchar() are buffered until RETURN is hit means getchar() does not see the characters until the user presses return. (i.e Enter Key)

Syntax for Accepting String and Working :

/* getchar accepts character & stores in ch */

```
char ch = getchar();
```

- When control is on above line then getchar() function will accept the single character. After accepting character control remains on the same line. When user presses the enter key then getchar() function will read the character and that character is assigned to the variable 'ch'.

Parameter	Explanation
Header File -	stdio.h
Return Type -	int (ASCII Value of the character)
Parameter -	Void
Use -	Accepting the Character

Example 1 :

In the following example we are just accepting the single character and printing it on the console –

```
main()
{
   char ch;
   ch = getchar();
   printf("Accepted Character : %c",ch);
}
```

Output :

Accepted Character : A

Example 2 : Accepting String (Use One of the Loop)

```
#include<stdio.h>
void main()
{
int i =0;
char name[20];
printf("\nEnter the Name : ");
while((name[i] =getchar())!='\n')
      i++ ;
      /*while loop will accept the one character at a time and check  it
with newline character. Whenever user enters newline character
then control comes out of the loop.*/
getch(); }
```

3.9.4 PUTCHAR():

Displaying String in C Programming

Syntax :

```
int putchar(int c);
```

Way 1 : Taking Character as Parameter

```
putchar('a') ;  //  Displays : a
```

- Individual Character is Given as parameter to this function.
- We have to explicitly mention Character.

Way 2 : Taking Variable as Parameter

```
putchar(a);
// Display Character Stored in a
```

- Input Parameter is Variable of Type“Character”.
- This type of putchar() displays character stored in variable.

Way3:Displaying Particular Character from Array

```
putchar(a[0]) ;
// Display a[0] th element from array
```

- Character Array or String consists of collection of characters.
- Like accessing individual array element , characters can be displayed one by one using putchar().

Example:

```
#include< stdio.h>
#include< conio.h>
int main()
{
  char string[] = "C programming\n";
  int i=0;
  while(string[i]!='\0')
  {
    putchar(string[i]);
    i++;
  }
return 0;
}
```

Output:

Cprogramming

3.9.5 PRINTF():

Syntax :

Way 1 : Messaging

```
printf (" Type your Message / Instruction " ) ;
```

Way 2 : Display String

```
printf ("Name of Person is %s", name ) ;
```

Notes or Facts :

printf is included in header file "**stdio.h**"

As name suggest it used for **Printing or Displaying Messages or Instructions**

Uses :

- Printing Message
- Ask user for entering the data (Labels . Instructions)
- Printing Results

3.10.1 ATOI FUNCTION:

- Atoi = Ato I = Alphabet to Integer
- **Convert String of number into Integer**

Syntax:

num = atoi(String);

num - Integer Variable
String- String of Numbers

Example :

```
char a[10] = "100";
int value = atoi(a);
printf("Value = %d\n", value);
return 0;
```

Output :
Value : 100

Significance :

- Can Convert any String of Number into Integer Value that can Perform the arithmetic Operations like integer
- Header File : **stdlib.h**

Ways of Using Atoi Function :

Way 1 : Passing Variable in Atoi Function

```
int num;
char marks[3] = "98";
num = atoi(marks);
printf("\nMarks : %d",num);
```

Way 2 : Passing Direct String in Atoi Function

```
int num;
num = atoi("98");
printf("\nMarks : %d",num);
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

OTHER INBUILT TYPECAST FUNCTIONS IN C PROGRAMMING LANGUAGE:

- Typecasting functions in C language performs data type conversion from one type to another.
- Click on each function name below for description and example programs.

  - atof() Converts string to float
  - atoi() Converts string to int
  - atol() Converts string to long
  - itoa() Converts int to string
  - ltoa() Converts long to string

3.10.2 STRLEN FUNCTION:

- **Finding length of string**

Point		Explanation
No of Parameters	-	1
Parameter Taken	-	Character Array Or String
Return Type	-	Integer
Description	-	Compute the Length of the String
Header file	-	string.h

Different Ways of Using strlen() :

- There are different ways of using strlen function. We can pass different parameters to strlen()function.

Way 1 : Taking String Variable as Parameter

```
char str[20];
int length ;
printf("\nEnter the String : ");
gets(str);
length = strlen(str);
printf("\nLength of String : %d", length);
```

Output:

Enter the String : hello

Length of String : 5

Way 2 : Taking String Variable which is Already Initialized using Pointer

```
char *str ="naresh";
int length ;
length = strlen(str);
printf("\nLength of String : %d", length);
```

Way 3 : Taking Direct String

```
int length ;
length = strlen("naresh");
printf("\nLength of String : %d",length);
```

Way 4 : Writing Function in printf Statement

```
char *str ="naresh";
printf("\nLength of String : %d",strlen(str));
```

3. *STRCAT FUNCTION:*

What strcat Actually does?

- Function takes 2 Strings / Character Array as Parameter
- Appends second string at the end of First String.

Parameter Taken - 2 Character Arrays / Strings

Return Type - Character Array / String

Syntax :

char* strlen (char * s1, char * s2);

Ways of Using Strcat Function :

Way 1 : Taking String Variable as Parameter

```
char str1[20] = "Don" , str2[20] = "Bosqo";
strcat(str1,str2);
puts(str1);
```

Way 2 : Taking String Variable which is Already Initialized using Pointer

```
char *str1 = "Ind",*str2 = "ia";
strcat(str1,str2);// Result stored in str1 puts(str1); // Result :
India
```

Way 3 : Writing Function in printf Statement

```
printf("nString: ", strcat("Ind","ia"));
```

4. *STRCMP FUNCTION:*

What strcmp Actually Does?

- Function takes two Strings asparameter.
- It returns integer.

Syntax :

```
int strcmp ( char *s1, char *s2 ) ;
```

Return Type		Condition
-ve Value	-	String1 < String2
+ve Value	-	String1 > String2
0 Value	-	String1 = String2

Example 1 : Two strings are Equal

```
char s1[10] = "SAM",s2[10]="SAM" ;
int len;
len = strcmp (s1,s2);
```

Output

0

/* *So the output will be 0. if u want to print the string then give condition like* */

```
char s1[10] = "SAM",s2[10]="SAM" ;
int len;
len = strcmp (s1,s2);
if (len ==0)
printf ("Two Strings are Equal");
```

Output:

Two Strings are Equal

Example 2 : String1 is Greater than String2

```
char s1[10] = "SAM",s2[10]="sam" ;
int len;
len = strcmp (s1,s2);
printf ("%d",len); //-ve value
```

Output:

-32

Reason :

ASCII value of “SAM” is smaller than “sam”

ASCII value of ‘S’ is smaller than ‘s’

Example 3 : String1 is Smaller than String1

```
char s1[10] = "sam",s2[10]="SAM" ;
int len;
len = strcmp (s1,s2);
printf ("%d",len); //+ve value
```

Output:

85

Reason :

ASCII value of “SAM” is greater than “sam”

ASCII value of ‘S’ is greater than ‘s’

1. *SPRINTF FUNCTION:*

- **sends formatted output to String.**

Features :

- Output is Written into String instead of Displaying it on the Output Devices.
- Return value is integer (i.e Number of characters actually placed in array / length of string).
- String is terminated by '\0'.
- Main Purpose : Sending Formatted output to String.
- Header File : **Stdio.h**

Syntax :

int sprintf(char *buf,char format,arg_list);

Example :

```
int age = 23 ;
char str[100];
sprintf( str , "My age is %d",age);
puts(str);
```

Output:

My age is 23

Analysis of Source Code: Just keep in mind that

- Assume that we are using printf then we get output “My age is 23”
- What does printf does ? —— Just Print the Result on the Screen
- Similarly Sprintf stores result “My age is 23” into string str instead of printing it.

3.11.2 SSCANF FUNCTION:

Syntax :

int sscanf(const char *buffer, const char *format[, address, ...]);

What it actually does?

- Data is read from array Pointed to by buffer rather than stdin.
- Return Type is Integer
- Return value is nothing but number of fields that were actually assigned a value

Example

```
#include <stdio.h>
int main ()
{
  char buffer[30]="Naresh 35";
  char name [20];
  int age;
  sscanf (buffer,"%s %d",name,&age);
  printf ("Name : %s\n Age : %d \n",name,age);
  return 0;
}
```

Output:

Name : Naresh

Age : 35

3. *STRSTR FUNCTION:*

- **Finds first occurrence of sub-string in other string**

Features :

- Finds the first occurrence of a sub string in another string
- Main Purpose : **Finding Substring**
- Header File : **String.h**
- Checks whether s2 is present in s1 or not
- On success, strstr returns a pointer to the element in s1 where s2 begins (points to s2 in s1).
- On error (if s2 does not occur in s1), strstr returns null.

Syntax :

char *strstr(const char *s1, const char *s2);

```
#include <stdio.h>
#include <string.h>
int main ()
{
  char string[55] ="This is a test string;
  char *p;
  p = strstr (string,"test");
  if(p)
  {
printf("string found\n");
```

```
printf("First string \"test\" in\"%s\" to  "\" \"%s \"" ,string, p);
  }
  else
          printf("string not found\n" );
   return 0;
}
```

Output:

string found

First string “test” in “This is a test string” to “test string”.

Example 2

Parameters as String initialized using Pointers

```
#include<stdio.h>
#include<string.h>
int main(void)
{
  char *str1 ="c4learn.blogspot.com", *str2 ="spot", *ptr;
  ptr =strstr(str1, str2);
  printf("The substring is: %sn", ptr);
  return 0;
}
```

Output :

The substring is: spot.com

Example 3
Passing Direct Strings

```
#include<stdio.h>
#include<string.h>
void main()
{
  char *ptr;
  ptr = strstr("c4learn.blogspot.com","spot");
  printf("The substring is: %sn", ptr);
}
```

Output :

The substring is: spot.com

3.11.4 STRREV():

- reverses a given string in Clanguage. Syntax for strrev( ) function is given below.

char *strrev(char *string);

- strrev() function is nonstandard function which may not available in standard library in C.

Algorithm to Reverse String in C:

- Start
- Take 2 Subscript Variables 'i','j'
- 'j' is Positioned on Last Character
- 'i' is positioned on first character
- str[i] is interchanged with str[j]
- Increment 'i'
- Decrement 'j'
- If 'i' > 'j' then goto step 3
- Stop

Example

```
#include<stdio.h>
#include<string.h>
int main()
{
  char name[30] = "Hello";
        printf("String before strrev() :%s\n",name);
        printf("String after strrev(%s", strrev(name));
        return 0;
}
```

Output:

```
String before strrev() : Hello
String after strrev()  : olleH
```

5. *STRCPY FUNCTION:*

Copy second string into First

What strcmp Actually Does?

- Function takes two Strings as parameter.
- Header File : String.h.
- It returns string.
- Purpose : Copies String2 into String1.
- Original contents of String1 will be lost.
- Original contents of String2 will remains as it is.

Syntax :

```
char*  strcpy ( char*string1, char*string2 );
```

- strcpy (str1, str2) – It copies contents of str2 into str1.
- strcpy (str2, str1) – It copies contents of str1 into str2.
- If destination string length is **less than source string**, **entire source string value won't be copied into destination** string.
- For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

Example 1

```
char s1[10] = "SAM" ;
char s2[10] = "MIKE" ;
strcpy (s1,s2);
puts (s1) ;      // Prints : MIKE
puts (s2) ;      // Prints : MIKE
```

Output:

```
MIK
E
MIKE
```

Example 2

```
#include <stdio.h>
#include <string.h>
int main( )
{
  char source[ ] = "hihello" ;
  char target[20]= "" ;
  printf ( "\nsource string = %s", source ) ;
  printf ( "\ntarget string = %s", target ) ;
  strcpy ( target, source ) ;
 printf("target string after strcpy()=%s",target);
   return 0; }
```

Output

```
source string = hihello
target string =
target string after strcpy( ) =hihello
```

6. *STRTOK FUNCTION*

- **tokenizes/parses the given string using delimiter.**

Syntax

char * strtok (char * str, const char * delimiters);

For example, we have a comma separated list of items from a file and we want individual items in an array.

- *Splits str[] according to given delimiters and returns next token.*
- *It needs to be called in a loop to get all tokens.*
- *It returns NULL when there are no more tokens.*

Example

```
#include <stdio.h>
#include <string.h>
int main()
{
        char str[] = "Problem_Solving_in_c";//Returns first token
        char* token = strtok(str, "_");//Keep printing tokens while
one of the delimiters present in str[].
        while (token != NULL){
                printf("%s\n", token);
```

```
token = strtok(NULL, "_");
        }
        return 0;
}
```

Output:

Problem
Solving
in
C

12. ARITHMETIC CHARACTERS ON STRING

- CProgramming Allows you to Manipulate on String
- Whenever the Character is variable is used in the expression then it is automatically Converted into Integer Value called ASCII value.
- All Characters can be Manipulated with that Integer Value.(Addition,Subtraction)

Examples :

ASCII value of : 'a' is 97

ASCII value of : 'z' is 121

Possible Ways of Manipulation :

Way 1:Displays ASCII value[Note that %d in Printf]

```
char x = 'a';
printf("%d",x); //  Display Result= 97
```

Way 2 :Displays Character value[Note that %c in Printf]

```
char x = 'a';
printf("%c",x); //  DisplayResult = a
```

Way 3 : Displays Next ASCII value[Note that %d in Printf]

```
char x = 'a' + 1 ;
printf("%d",x); //Display Result = 98 (ascii of 'b' )
```

Way 4 Displays Next Character value[Note that %cin Printf]

```
char x = 'a' + 1;
printf("%c",x); //  Display Result= 'b‘
```

Way 5 : Displays Difference between 2 ASCII in Integer[Note %d in Printf]

```
char x = 'z' - 'a';
printf("%d",x);/*Display Result = 25 (difference between ASCII of z and a ) */
```

Way 6 : Displays Difference between 2 ASCII in Char [Note that %cin Printf]

```
char x = 'z' - 'a';
printf("%c",x);/*Display Result =( difference between ASCII of z and a ) */
```

13. FUNCTION DECLARATION AND DEFINITION:

- A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
- The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.
- A function can also be referred as a method or a sub-routine or a procedure, etc.

3.13.1 Defining a function

The general form of a function definition in C programming language is as follows −

```
return_type function_name( parameter list )
{
body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

Return Type − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

```
main()
{
display();
}
void mumbai()
{
printf("In mumbai");
}
void pune()
{
india();
}
void display()
{
pune();
}
void india()
{
mumbai();
}
```

We have written functions in the above specified sequence , however functions are called in which order we call them.

Here **functions are called this sequence** –

main()
display()
pune()
india()
mumbai().

Why Funtion is used???

Advantages of Writing Function in CProgramming

1. Modular and Structural Programming can be done

- We can divide c program in **smaller modules**.
- We can call module whenever require. e.g suppose we have written calculator program then we can write 4 modules (i.e add,sub,multiply,divide)
- Modular programming **makes Cprogram more readable**.
- Modules once created , **can be re-used in other programs**.

2. It follows Top-Down Execution approach , Somain can be kept very small.

- Every Cprogram starts **from main function**.
- Every function is **called directly or indirectly through main**
- Example : **Top down approach**. (functions are executed from top to bottom)

3. Individual functions can be easily built,tested

- As we have developed Capplication in modules **we can test each and every module**.
- **Unit testing** is possible.
- Writing code in function will **enhance application development process**.

3. Program development become easy

4. Frequently used functions can be put together in the customized library

- We can put frequently used functions in **our custom header file**.
- After creating header file we can re use header file. We can include header file in other program.

5. A function can call other functions & also itself

- Function can call other function.
- Function can call itself , which is called as "recursive" function.
- Recursive functions are also useful in order to write system functions.

6. It is easier to understand the Program topic

- We can get overall idea of the project just by reviewing function names.

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

How Function works in CProgramming?

- Cprogramming is modular programming language.
- We must divide Cprogram in the different modules in order to create more readable, eye catching ,effective, optimized code.
- In this article we are going to see how function is Cprogramming works ?

Explanation : How function works in C Programming ?

- Firstly Operating System will call our main function.
- When control comes inside main function , execution of main starts (i.e execution of Cprogram starts)
- Consider Line 4 :

  num = square(4);
- We have called a function square(4). [See : How to call a function ?].
- We have passed "4" as parameter to function.

Note : Calling a function halts execution of the current function , it will execute called function

- after execution control returned back to the calling function.
- Function will return 16 to the calling function.(i.e. main)
- Returned value will be copied into variable.
- printf will gets executed.
- main function ends.
- Cprogram terminates.

FUNCTION PROTOTYPE DECLARATION IN C PROGRAMMING

- Function prototype declaration is necessary in order to provide information the compiler about function, about return type, parameter list and function name etc.

Important Points :

- Our program starts from main function. Each and every function is called directly or indirectly through main function
- Like variable we also need to declare function before using it in program.
- In C, declaration of function is called as prototype declaration
- Function declaration is also called as function prototype

Points to remember

- Below are some of the important notable things related to prototype declaration –
- It tells name of function, return type of function and argument list related information to the compiler
- Prototype declaration always ends with semicolon.
- Parameter list is optional.
- Default return type is integer.

Header Files
Global Declaration Section
(Write Prototype Declaration)
Write Main Function in this section
www.c4learn.com
Write All Function Definitions in this Section

Syntax

return_type function_name (type arg1, type arg2.....);

prototype declaration comprised of three parts i.e name of the function,return type and parameter list

Examples of prototype declaration

- Function with two integer arguments and integer as return type is represented using syntax **int sum(int,int);**
- Function with integer argument and integer as return type is represented using syntax **int square(int);**
- In the below example we have written function with no argument and no return type **void display(void);**
- In below example we have delclared function with no argument and integer as return type **int getValue(void);**

Positioning function declaration

- If function definition is written after main then and then only we write prototype declaration in global declaration section
- If function definition is written above the main function then ,no need to write prototype declaration

Case 1 : Function definition written before main

```
#include<stdio.h>
void displayMessage()
{
printf("welcome");
}
void main()
{
displayMessage();
}
```

Case 2 : Function definition written after main

```
#include<stdio.h> //Prototype Declaration
void displayMessage();
void main()
{
displayMessage();
}
void displayMessage()
{
printf("welcome");
}
```

Need of prototype declaration

- Program Execution always starts from main , but during lexical analysis (1st Phase of Compiler) token generation starts from left to right and from top to bottom.
- During code generation phase of compiler it may face issue of backward reference.

14. TYPES OF CALLING

- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.
- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

```
#include <stdio.h>
/*  function declaration */
int max(int num1, int num2);
int main ()
{
/* local variable definition */
 int a = 100;
 int b =200;
 int ret;    /* calling a function to get max value */
 ret = max(a, b);
printf( "Max value is : %d\n", ret );
return 0;
} /* function returning the max between two numbers */
int max(int num1, int num2)
{   /* local variable declaration */
int result;
if (num1 >num2)
result = num1;
else
result = num2;
return result;  }
```

output

Max value is : 200

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways in which arguments can be passed to a function −
- By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

Sr.No.	Call Type & Description

1	**Call by value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
	/* function definition to swap the values */
	void swap (int x, int y)
	{
	int temp; temp =x;
	/* save the value of x */
	x = y; /* put y into x */
	y = temp; /* put temp into y */
	return; }

2	**Call by reference** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
	/* function definition to swap the values */
	void swap (int x, int y)
	{
	int temp;
	temp =*x;
	*x =*y;
	*y =temp;
	return; }

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

1. CALL BY VALUE

- While Passing Parameters using call by value , xerox copy of original parameter is created and passed to the called function.
- Any update made inside method will not affect the original value of variable in calling function.
- In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively.
- As their scope is limited to only function so they cannot alter the values inside main function.

```
#include <stdio.h>
void swap(int x, int y); /* function declaration */
int main ()
{
int a = 100; /* local variable definition */
int b =200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );/*calling a function to swap the values */
swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Output:

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200

3.14.2 CALL BY REFERENCE

- The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

- To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
#include <stdio.h>
void swap(int *x, int *y); /* function declaration */
int main ()
{
int a = 100; /* local variable definition */
int b = 200;
printf("Before swap, value of a : %d\n", a);
printf("Before swap, value of b : %d\n", b ); /*calling a function to swap the values. * &a indicates pointer to a ie. address of variable a and * &b indicates pointer to b ie. address of variable b.*/
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

output

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

15. .1 FUNCTION WITH ARGUMENTS AND NO RETURN VALUE :

- Function accepts argument but it does not return a value back to the calling Program .
- It is Single (One-way) Type Communication
- Generally Output is printed in the Called function

Function declaration : void function();

Function call : function();

Function definition : void function() { statements; }

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);
int main()
{
int n; printf("Enter a positive integer: ");
scanf("%d",&n);
// n is passed to the function checkPrimeAndDisplay(n);
return 0;
}
```

```
// void indicates that no value is returned from the function
void checkPrimeAndDisplay(int n)
{
int i, flag =0;
for(i=2; i <= n/2; ++i)
{
if(n%i == 0){
flag = 1;
break; }
}
if(flag == 1)
printf("%d is not a prime number.",n);
else
printf("%d is a prime number.", n);
}
```

OUTPUT

Enter a positive integer : 4

4 is not a prime number

3.15.2 FUNCTION WITH NO ARGUMENTS AND NO RETURN VALUE IN C

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

Function declaration : void function();

Function call : function();

Function definition : void function()

{

statements;

}

```
#include<stdio.h>
void area(); //  Prototype Declaration
void main()
{
area();
}
```

```
void area()
{
   float area_circle;
   float rad;
    printf("\nEnter the radius : ");
   scanf("%f",&rad);
    area_circle = 3.14 *  rad *  rad ;
    printf("Area of Circle = %f",area_circle);
}
```

Output :

Enter the radius : 3

Area of Circle = 28.260000

3.16.1 FUNCTION WITHOUT ARGUMENTS AND RETURN VALUE

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. Aexample for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.

Function declaration : int function();

Function call : function();

Function definition : int function() { statements; return x;}

```
#include<stdio.h>
 int sum();
 int main()
{
   int addition;
   addition = sum();
   printf("\nSum of two given values = %d", addition);
```

```
 return 0;
}

int sum()
{
   int a = 50, b = 80, sum;
   sum = a + b;
   return sum;
}
```

OUTPUT

Sum of two given values = 130

3.16.2 FUNCTION WITH ARGUMENTS AND RETURN VALUE

Syntax :

Function declaration : int function (int);

Function call : function(x);

Function definition: int function(int x) { statements; return x; }

```
#include<stdio.h>

float calculate_area(int);

int main()

 {

int radius;

float area;

printf("\nEnter the radius of the circle : ");

scanf("%d",&radius);

area = calculate_area(radius);

printf("\nArea of Circle : %f",area);
```

```
return(0);
}
float calculate_area(int radius)
{
float areaOfCircle;
areaOfCircle = 3.14 *  radius * radius;
return(areaOfCircle);
}
```

Output:

Enter the radius of the circle : 2

Area of Circle : 12.56

17. PASSING ARRAY TO FUNCTION IN C

Array Definition :

Array is collection of elements of similar data types .

Passing array to function :

Array can be passed to function by two ways :

- **Pass Entire array**
- **Pass Array element by element**

1 . Pass Entire array

- Here entire array can be passed as a argument to function .
- Function gets **complete access** to the original array .
- While passing entire array Address of first element is passed to function , any changes made inside function , directly **affects the Original value** .
- Function Passing method : "**Pass by Address**"

2 . Pass Array element by element:

- Here individual elements are passed to function as argument.
- Duplicate **carbon copy of Original variable** is passed to function .
- So any changes made inside function **does not affects the original value**.
- Function doesn't get complete access to the original array element.
- Function passing method is "**Pass by Value**".

Passing entire array to function :

- Parameter Passing Scheme : **Pass by Reference**
- Pass name of array as function parameter.
- Name contains the base address i.e (Address of 0th element)
- Array values are updated in function .
- Values are reflected inside main function also.

```
#include<stdio.h>
#include<conio.h>
void fun(int arr[])
{
int i;
for(i=0;i< 5;i++)
arr[i] = arr[i] + 10;
}
void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
```

```
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing entire array .....");
fun(arr); // Pass only name of array
for(i=0;i< 5;i++)
printf("\nAfter Function call a[%d] : %d",i,arr[i]);
getch();
}
```

output

```
Enter the array elements : 1 2 3 4 5
Passing entire array .....
After Function call a[0] : 11
After Function call a[1] : 12
After Function call a[2] : 13
After Function call a[3] : 14
After Function call a[4] : 15
```

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

Passing Entire 1-D Array to Function in CProgramming

- Array is passed to function Completely.
- Parameter Passing Method : **Pass by Reference**
- It is Also Called "**Pass by Address**"
- Original Copy is Passed to Function
- Function Body Can Modify **Original Value**.

Example :

```
#include<stdio.h>
#include<conio.h>
void modify(int b[3]);
void main()
{
int arr[3] ={1,2,3};
modify(arr);
for(i=0;i<3;i++)
printf("%d",arr[i]);
getch();
}
void modify(int a[3])
{
int i;
for(i=0;i<3;i++)
a[i] =a[i]*a[i];
 }
```

Output :

1 4 9

Here **"arr"** is same as **"a"** because Base Address of Array **"arr"** is stored in Array **"a"**

Alternate Way of Writing Function Header :

void modify(int a[3]) **OR** void modify(int *a)

Passing array element by element to function :

- Individual element is passed to function using **Pass By Value** parameter passing scheme
- Original Array elements remains same as Actual Element is never Passed to Function. thus function body cannot modify **Original Value**.
- Suppose we have declared an array 'arr[5]' then its individual elements are arr[0],arr[1]…arr[4]. Thus we need 5 function calls to pass complete array to a function.

Tabular Explanation :

Consider following array

Iteration	Element Passed to Function	Value of Element
1	arr[0]	11
2	arr[1]	22
3	arr[2]	33
4	arr[3]	44
5	arr[4]	55

C Program to Pass Array to Function Element by Element :

```
#include<  stdio.h>
#include< conio.h>
void fun(int num)
{
printf("\nElement : %d",num);
 }
 void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing array element by element.....");
for(i=0;i< 5;i++)
fun(arr[i]);
getch();  }
```

Output :

Enter the array elements : 1 2 3 4 5
Passing array element by element.....
Element : 1
Element : 2
Element : 3
Element : 4
Element : 5

DISADVANTAGE OF THIS SCHEME :

- This type of scheme in which we are calling the function again and again but with **different array element is too much time consuming**. In this scheme we need to call function by pushing the current status into the system stack.
- It is better to pass complete array to the function so that we can save some system time required for pushing and popping.

18. RECURSION CONCEPT :

- Recursion is **basic concept** in Programming.
- When Function is defined in terms of itself then it is called as "**Recursion Function**".
- Recursive function is a **function which contains a call to itself**.

```
int factorial(int n)
{
if(n==0)
return(1);
else
return( n * factorial(n-1)):
}
```

$$\text{Factorial}(n) = \begin{cases} 1 \\ n * \text{Factorial}(n-1) \end{cases}$$

Example:

```
#include <stdio.h>
int sum(int n);
int main()
{
int number, result;
printf("Enter a positive integer: ");
scanf("%d", &number);
result = sum(number);
printf("sum=%d", result);
}
int sum(int num)
{
if (num!=0)
return num + sum(num-1); //  sum() function calls itself
else
return num;
}
```

Output

Enter a positive integer: 3 6

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

How does recursion work?

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

recursive call

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY, CHENNAI.

```
int main() {
... .. ...
result = sum(number)
... .. ...
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}
```

3 3 2 2 1 1 0

3+3 = 6
is returned

1+2 = 3
is returned

0+1 = 1
is returned

0
is returned

Warning of using recursive function in CProgramming :

- Recursive function must have at least one terminating condition that can be satisfied.
- Otherwise, the recursive function will call itself repeatably until the run time stack overflows.

Advantages and Disadvantages of Recursion

- Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.
- If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow. Instead, you can use loop.

Thank you

Unit-3

Conditional Program Execution

Normally The statements in a program are executed in the order in which they appear in program. This type of execution is called sequential execution

→ Some times it is necessary to

* Select a set of statements for several alternatives.
* Skip certain statements depending on some conditions in the programs and continue the execution from some other point.
* Repeat a set of statements for a known number of times or untill a specified condition is fulfilled.

Condition Control Statements

* Some time it is necessary to check the condition to make the decision.
* This involves performing a logical test, the result

* Depending upon the condition of given statement which statement will execute next is determined.

* After that, the control is transferred to that statement & starts execution from that point.

* This entire process is known as conditional execution which includes both decision making & branching.

If Statement:

* Using If statement the logical condition is tested which results in either true or false.

* If the result is true (nonzero) then statements that immediately follow "If" is executed.

* If the result is false (zero) then control transfers to next executable statement, that immediately follow it.

→ Syntax of writing (if)

```
if ( Condition)
{
  statement
  - - -
}
```

where:

Condition → logical expression that result in true or false

Statement → simple or compound statement

<u>Note:</u> for simple (or single statement) no braces are required.

<u>Prog</u>

```
main ()
{ int num;
printf ("Enter a number\n");
scanf ("%d", &num);
if (num == 0)
```

```
printf("You entered zero");
}
```

Prog

```
main()
{
    float x, y, ratio;
    printf("Enter X & Y \n");
    scanf("%f %f", &x, &y);
    ratio = x/y;
    if (ratio > 0)
    {
        printf("Ratio is greater than zero\n");
        printf("You are using if condition\n");
    }
    printf("Ratio is less than or equal to zero");
}
```

If-Else Statement

* If statement is used to execute only one action, If there are two statements to be executed alternatively then if-else statement is used.

* The if-else statement is a two way branching.

⇒ Syntax

```
if (condition)
  {
    statement1;
  }
else
  {
    statement2;
  }
```

* The condition is tested and if the result of this logical test is true, then statement 1 is executed otherwise statement2 is executed.

* After executing one of these two statement, the control transfers to the statement immediately following if-else structure.

Prog

```
main()
{
   int m, n;
   printf ("Enter the value of M & N\n");
   scanf ("%d %d", &m, &n);
   if (m==n)
      { printf (" Both numbers are equal\n");
      }
else
      {
        printf ("Numbers are not equal\n");
      }
}
```

Output

Case 1

Enter value of M & N
26
26
Both numbers are equal

Case 2

Enter value of M & N
34
44
Both numbers are not equal.

Prog

```
main()
{ char ch;
  printf("Enter a character\n");
  scanf("%c", &ch);
  if (ch=='y' || ch=='Y')
    printf("Yes");
  else
      printf("No");
}
```

Nested If statement

* If there are more than two alternatives to select, then the nested-if statements are used.
* Enclosing if within another if is called a nested if statement.

Syntax

```
if (condition1)
{
    if (condition2)
    {
      statement1;
    }
    else
    {
      statement2;
    }
}
else if (condition3)
    {
      statement3;
    }
else {
      statement4;
}
```

Program to find largest number out of given 3 Nos

```
main ( )
{ int a, b, c;
  printf (" Enter the value of A, B, C \n");
  scanf ("%d %d %d", &a, &b, &c);
  if (a>b)
   {
     if (a>c)
     printf (" A is largest");
```

```
else
    printf ("C is largest\n");
}
if (b>c)
    printf ("B is largest");
else
    printf ("C is largest");
}
```

Q. Write a Program to accept two integers as coordinate of point & determine its quadrant

```
# include <stdio.h>
# include <conio.h>
void main()
{   int x, y;
    printf ("Enter the value of x, y for a point\n");
    scanf ("%d %d", &x, &y);
    if (x>0 && y>0)
        printf (" Point lies in first quadrant\n");
    if (x>0 && y<0)
        printf (" Point lies in fourth quadrant\n");
```

```
if (x<0 && y>0)
    printf ("Point lies in second quadrant\n");
if (x<0 && y<0)
    printf ("Point lies in Third quadrant\n");
if (x==0 && y==0)
    printf ("Point lies at origin");
}
```

Darling Else Problem :

In nesting if-else statement, there exists a problem when there is no matching else for every if. This problem is known as Darling else problem.

eg

```
if (condition1)
    if (condition2)
        statement1;
    else
        Statement2
```

Solution

```
if (expression1)
{
    if (expression2)
        statement 1;
}
else
    statement 2;
```

} body of if

} body of else

The If-Else if ladder

* To make multiway decision based on several condition if-else if ladder is used.

* As soon as one of the given if or else if gives a true result, the following statement or block is executed, & no further comparisions are performed.

* If none of the condition is true, the final else is executed

* Final else is optional, if final else is not present no action took place even if all the condition are false.

Syntax

```
if (condition)
    statement;
else if ( condition)
    statement;
else if (condition)
    statement;
else
    statement;
```

Prog

```
if (result>=75)
    printf (" Grade A \n");
else if ( result >=60)
    printf (" Grade B \n");
else if ( result >= 40)
    printf ("Grade C \n");
else
    printf (" Fail");
```

Q. Write a program in C to test whether the given year is leap year or not.

Prog

```
# include < stdio.h>
# include < conio.h>
void main()
  { clrscr();
    int year;
    printf (" Enter the year to be checked\n);
    scanf ("%d", &year);
    if (year % 4 == 0)
      { printf ("Year is leap");
      }
    else
      { printf (" Year is not a leap year");
      }
    getch();
  }
```

Q. Write a program to check whether a given number is "odd" or "even".

Program

```
# include < stdio.h>
# include < conio.h>
Void main()
  {
    clrscr();
    int n;
    printf ("Enter the number");
    scanf ("%d", &n);
  if (n%2==0)
    {
      printf ("Number is even");
    }
  if (n%2!==0)
    {
      printf ("Number is Odd");
    }
  getch();
}
```

Q. Calculation of Area of Circle or Rectangle depending upon choice of user (by entering 'C' or 'R')

Program

```
# include < stdio.h >
# include < conio.h >
Void main ( )
    {
      Char ch;
      float a1, a2, area;
    printf ("Enter C for area of circle OR
                R for area of Rectangle");
    scanf ("%c", &ch);
    if (ch == 'C' || ch == 'c').
    {
      printf ("Enter Radius of circle\n");
      scanf ("%f", &a1);
      area = 3.14 * a1 * a1;
      printf ("The area of circle is %f", area);
    }
    if (ch == 'R' || ch == 'r')
```

```
{
printf("Enter length & width : ");
scanf("%f %f", &a1, &a2);
area = a1 * a2;
printf("Area of rectangle is %f", area);
}
}
```

Switch Statement :

* Switch statement is a multi-way decision that tests whether an expression matches one of the a number of constant integer values and branches accordingly.

* This is another form of multiway decision.

Syntax

```
Switch (expression)
{
  Case Value1 :
        statements
        break;
  Case Value2 : statements
```

```
Case Value3:
            statement
            break;
    default:
            statement
}
```

* The keyword switch is followed by switch variable or expression in parenthesis
* The case keyword is followed by a constant, followed by colon.
* When switch statement is executed, the expression is evaluated and its value is successively compared with case constants.
* When a match is found, the statement sequence associated that case is executed untill the break statement or end of switch is reached
* Default statement is executed if no match is found

Prog

```
# include <stdio.h>
void main( )
{
  int n1, n2, op;
  printf ("Enter two numbers");
  scanf ("%d %d", &n1, &n2);

  printf ("\n Menu");
  printf ("\n 1. Addition");
  printf ("\n 2. Substraction");
  printf ("\n 3. multiplication");
  printf ("\n 4. Division");
  printf ("\n Select your choice by entering
          a Number");
  scanf ("%d", &op);

  switch (op)
  {
   case 1:
          printf ("\n Addition is %d", n1+n2);
          break;
  case 2:
       printf ("\n Difference is %d", n1-n2);
          break;
```

```
Case 3:
        printf ("\nMultiplication is %d", n1*n2);
        break;
Case 4:
        printf ("\nQuotient is %d", n1/n2);
        printf ("\n Remainder is %d", n1%n2);
  }
}
```

Q. Write a program that calculate area of Circle, Rectangle, triangle, square by displaying Menu.

Program

```
# include <stdio.h>
# include <conio.h>
void main ()
    { clrscr();
      int code;
    float side, base, height, length, breadth,
      area, radius;

printf ("menu\n");
```

```
printf ("\n 1. Circle");
printf ("\n 2. Square");
printf ("\n 3. Triangle");
printf ("\n 4. Rectangle");
printf ("\nEnter your choice by entering number");
scanf ("%d" & code);
    switch (code)
      Case 1 :
            printf ("\n Enter Radius of Circle");
            scanf ("%f", & radius);
            area = 3.14 * radius * radius;
        printf ("\n the area of circle is %f", area);
            break;
      Case 2 :
            printf ("\n Enter side of square");
            scanf ("%f", & side);
            area = side * side;
        printf ("\n the area of square is %f", area);
            break;
```

```
Case 3 :
        printf ("\n Enter the Base & height of
                                triangle ");
        scanf ("%f %f", & base, & height);
        area = 0.5 * base * height;
        printf ("\n Area of triangle is %f", area);
            break;

Case 4 :
        printf ("\n Enter length & breadth of Rectangle");
        scanf ("%f %f", & length, & breadth);
        area = length * breadth;
        printf ("\n Area of Rectangle is %f", area);
            break;

    default : printf ("\n Error in code");
}

}
```

Q. Write a program to accept a letter & print grade as per given table:

Letter	Remark
S	Super
A	Very Good
B	Fair
Y	Absent
F	Fail

Prog

```
# include <stdio.h>
# include <conio.h>
Void main()
    {
      Char letter;
      printf ("Enter the letter\n");
      scanf ("%c", & letter);
    switch (letter)
    {
      case 'S' :
            printf ("Super\n");
            break;
```

```
Case 'A' :
        printf ("Very Good \n");
          break;
Case 'B' :
        printf (" Fair\n);
          break;

 Case 'Y' :
        printf (" Absent \n");
            break;
  Case 'F':
        printf (" Fail \n");
          break;
  default :
        printf (" Wrong choice");
  }

}
```

GOTO Statement

⇒ C Language supports unconditional control statement "goto" to transfer the control from one point to another in program.

Syntax

```
goto label;
```

where goto → a keyword
label → a symbolic constant where the control will transfer

Note: A label can be placed anywhere in the C Program either before or after goto statement.

eg

```
goto END;
  ____
  ____
  ____
  ____
END:
  ____
  ____
  ____
```

```
START:
  ____
  ____
  ____
goto START;
```

Program

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

void main ( )
  {
    int goals;
  printf (" Enter Number of goals \n");
  scanf ("%d", &goals);
  if ( goals <= 5 )
    goto SOS;
  else
    {
      printf (" Good Player");
    }
SOS:
    printf (" Try Your Best");
}
```

Looping

⇒ Looping is a powerful programming technique through which a group of statements are executed repeatedly, untill a specific condition is satisfied

⇒ Looping is also called a repetitive or an iterative control mechanism.

⇒ Loop consists of two parts, one is called the body of loop and other is called control statement.

⇒ Control statement perform logical test whose result is either true or false. If the result of this test is true, then the body of loop will execute otherwise loop terminates.

⇒ If control statement is placed before the body of loop, it is called entry control loop & if control statements are at end of ~~loop~~ (body) it is called exit controlled loop.

⇒ Three efficient looping mechanism are:

① <u>Initialisation</u>: To set initial value of loop counter

② <u>Decision</u>: A test condition to decide whether loop will execute or not

③ <u>Updation</u>: Incrementing or decrementing counter value.

<u>Examples of loops</u>

① While statement

② do-while statement

③ for statement

<u>While Loop</u>

While loop executes untill the condition specified in while becomes false

Syntax

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

where

while → is a keyword

expression → logical expression that results either true or false

Statements → statement which executes for while

* In while loop, firstly the logical expression is tested and if the result is true, the statements in while body will executes, this entire process continues untill the logical expression become false.

* This is also called as pretested loop.

* The value of loop counter will change during each pass of while loop.

<u>Note</u>: If the value of loop counter doesn't changes in each pass, the loop will enters into infinite loop condition

Program to print your name in desired number of times

```
# include <stdio.h>
void main( )
  {
    int R;
    int i;
    printf ("How many times you want to print your name\n");
    scanf ("%d", & R);
    i=0;
    while ( i< R)
    {
      printf ("Ashish Chauhan\n");
    }  i++;
```

Q. Program to find sum of first 'n' numbers

Prog

```
# include <stdio.h>
# include <conio.h>
void main ( )
    {
      clrscr ( );
      int n, i, sum=0;
     printf ("Enter the number \n");
      scanf ("%d", & n)
       i= 1;
      while (i<=n)
       {
          sum += i;
           i++;
       }
      printf (" The sum of given number is %d", sum)
    }
```

Q. Factorial of number

```
while (num > 0)
    {
      ans = ans * num;
      num-- ;
    }
```

Do while loop

⇒ Do while loop is used to execute a set of statements repeatedly, untill the logical test result becomes false.

⇒ This is called post-test loop because the test is made at the end of each pass

Syntax

```
do
{
    statements
}
while (expression);
```

Note:

(i) The body of do while will execute atleast once because the logical test is carried out at the end of pass.

(ii) The body of do while must contain statement to modify the variable tested in the logical expression condition of while.

Q. Program to find sum of all odd numbers between 1 to 50

```
# include < stdio.h >
void main( )
  {
    int odd_num=1, sum=0;
    do {
        sum+ = odd_num;
        odd_num+=2;
      }
      while (odd_num<=50);
```

```
printf ("Sum = %d \n", sum);
}
```

Q0 Print Colour Code of Rainbow as long as user wants

```
# include <stdio.h>
# include <conio.h>
void main()
    {
        clrscr();
        int color-code, choice;
    do
      {  printf ("Enter Color Code \n");
         scanf ("%d", & color- code);
    switch (color- code)
      {
        case 1:  printf ("Color is voilet \n");
                 break;
        case 2:  printf (" Color is Indigo \n");
                 break;
        case 3:  printf ("Color is Blue \n");
                 break;
```

```
Case 4:
        printf ("Color is Green\n");
            break;
Case 5:
        printf ("Color is Yellow\n");
            break;
Case 6:
        printf ("Color is orange\n");
            break;
Case 7:
        printf ("Color is Red\n");
            break;
default:    printf ("Rainbow has only 7 colors\n");
}

printf ("Do you want to continue if yes press1 if
        no print 0 \n");
scanf ("%d", &choice);
}
  while (choice==1);
}
```

Program to find sum of series

$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7}$ ----- upto n terms

```
# include <stdio.h>
# include <conio.h>
void main ()
  {
    int n, sign, const;
    float sum=0, term, num, den;
    printf ("Enter the number of terms in series\n"),
    scanf ("%d", &n);
    sign = 1;
    num = 1;
    den = 1;
    Const = 1;
   do
    {
      term = (num/den) * sign;
      sum = sum + term;
      den++;
      sign *= (-1);
    }  Const++;
```

```
while (const <= n);
printf ("Sum of series = %of \n", sum);
}
```

For Loop

⇒ For loop is used when the programmer knows how many times a set of statements are to be executed.

Syntax

```
for (expression1 ; expression2 ; expression3)
{
    statements ;
}
```

where:

expression1 → initialises the loop index before loop begins

expression2 → Conditional expression that tests whether the loop will continue or stops

expression3 → modifies loop index after each iteration

statements → body of loop which executes for loop

Q. Program to print first 10 numbers using FOR

```
# include < stdio.h>
# include < conio.h>
Void main ()
{
    int n, i;
    for( i=0; i<=10; i++)
    {   printf("\n");
        printf ("%d", i);
    }
}
```

Also see

```
# include < stdio.h>
Void main()
{ int i=0, limit=5;
  for ( ; i< limit; )
  {   i++;
      printf("%d\n", i);
  }
}
```

Q. Write a program to calculate the sum of fibonacci series upto 100 terms.

```
# include < stdio.h >
# include < conio.h >
void main()
{
    int i ;
    int fib1= 0 , fib2 = 1, fib3 , count = 0;
printf ("%d", fib1);
printf ("%d", fib2);
for (i=0 ; i<=100; i++)
    {
        fib3 = fib1 + fib2;
        printf ("%d", fib3);
        fib1 = fib2;
        fib2 = fib3;
    }
```

Nested For Statement

⇒ In nested "for" there are two for statements. The first one is called outer loop & the second one is the inner loop.

⇒ The statement1 & statement2 are repeatedly execute the number of times specified by the maximum limit of the second for loop.

```
for (exp11; exp12; exp13)
  {
    for (exp21; exp22; exp23)
      {
        Statement 1;
        Statement 2;
      }
  }
```

Prog To print.

```
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

Sol^n

```
# include <stdio.h>
void main ()
{
    int i, j, num=5;
    printf(" Number Pattern \n");
    for (i=num ; i>1; i--)
    {
        for (j=1 ; j<=i ; j++)
        {
            printf("%d", num);
        }
        printf ("\n");
    } num--

}
```

Program to print following Pattern

```
*
* *
* * *
* * * *
* * * * *
```

```
# include < stdio.h >
void main ()
  {
    int i, j;
    Char ch = '*';
  for (i=0; i<5; i++)
    {  for (j=0; j<=i; j++)
       {
         printf ("%c", ch);
       }
       printf ("\n");
    }

  }
```

Jump in loop

Break Statement:

→ Break statement allows to terminate the execution of loop & jump out of the loop, without getting back to conditional test.

→ when break statement is encountered inside a loop, the loop is immediately terminated & control of execution is ~~ter~~ transferred to next statement immediately following the loop.

eg

```
for (i=0; i<100; i++)
  {
    printf("%d", i);
    if (i==10)
      break;
  }
_____
_____ next statement
_____
```

Program to check the sum of first 'n' numbers & if the sum is greater than 20 print the sum

```
# include <stdio.h>
# include <conio.h>
void main()
  {
     int sum=0, n, i ;
    printf ("Enter the limit \n");
    scanf ("%d", &n);
    for (i=0; i<n; i++)
     {
       sum+=i ;
        if (sum >20)
          break;
     }
    printf ("The sum is %d", sum);
  }
```

Continue Statement :

→ Continue statement is used for the next iteration of the loop to take place, skipping any code in between.

→ It is used to terminate the current iteration & continue with the next iteration of the loop.

→ Continue statement causes the updation & then the conditional test portion of loop to execute.

eg

```
for( i=1; i<=2; i++)
  {
    for (j=1; j<=2; j++)
      {
        if (i==j)
          continue;
        printf ("%d %d\n", i, j);
      }
  }
```

output

```
12
21
```

```
# include < stdio.h>
void main( )
  {
    int kout; sum1 = 0, sum2 = 0, limit, rem;
    printf (" Enter limit\n");
    scanf ("%d", & limit);
    for (kout = 0 ; kout < limit ; kout++)
      {
        rem = (kout % 2) ;
        if (rem = = 0)
        {
          sum1 + = kout ;
        } continue ;
        sum2 + = kout ;
      }
    printf (" Sum1 = %d\n", sum1);
    printf (" Sum2 = %d\n", sum2);
  }
```

Functions :

* A function is a set of instructions to carry out a particular task.

* Generally the functions are classified into standard function & user defined functions.

* sqrt(), pow(), sin() etc are standard functions, such functions are selective.

* Programmer can define other functions also and are named as user defined functions.

Defination of Function

```
type name_of_function (parameters list)
{
    Variable declaration ;
    statement 1
    :
    return (value_computed)
}
```

where

type → data type of value returned by function

name-of-function → name of user defined function

parameter list → list of variables that receives value from calling function

return → a keyword used to send output of function, back to the calling function.

value-computed → The value to be returned to the calling function after processing

Function Basics

1) Function Declaration: All functions in C language need to be declared before they are used.

2) Function definition: It is actual code for the function to perform specific task.

③ <u>Function Call:</u> In order to use function in program, function call is used. The program or function that calls the function is reffered as calling function

<u>ex</u>

```
# include <stdio.h>
void print_message();   // function declaration
void main()
  {
    print_message();  // function call
  }

print_message()  // function definition
  {
  printf("This is a module called print_message\n");
  }
```

Program to add two numbers & display result using function addnum().

Sol^n

```
# include < stdio.h >
Void main()
  {
   int n1, n2, result;
   printf ("Enter two numbers \n");
    scanf ("%d %d", &n1, &n2);
   result = addnum (n1, n2) ;
    printf ("The sum of two numbers is %d", result);
  }
  int addnum (int Val1, int Val2)
   {
     int sum;
     sum = Val1 + Val2;
      return (sum);
   }
```

Arguments & Parameters

Arguments and parameters are variables used in a function.

→ Variables used in a function reference are called arguments. These are written in paranthesis followed by the name of function in function call.

→ Variables used in the function definition are called parameters. They are also referred to as formal parameters, because they are not accepted values

These are written in paranthesis followed by name of function in function definition.

Function with Arguments but No Return Value

→ Here the called function receives the data from the calling function.

* The arguments and parameters should match in number, data type & order, but the called function does not return any value back to the calling function.

* It is one way data communication between the calling function & called function

Scope Rule:

Scope Rule explains, if an entity (ie variable, parameter and function) is visible or accessible of certain places, Thus, places where an entity can be accessed or visible is reffered to the scope of that entity.

The simplest rule is as follows

1. The scope of an entity is the program or function in which it is declared.
2. A global entity is visible to all contained functions. Including the function in which that entity is declared
3. An entity declared in the scope of another entity is always a different entity if their names are identical.

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

void main()
  {
    int x, y;
    printf("Enter the value of x & y\n");
    scanf("%d %d", &x, &y);
    maximum (x, y);
  }

~~maximum (p, q)~~
maximum (int p, int q)
  {
    if (p>q)
      printf("Maximum = %d", p);
    else
      printf("maximum = %d", q);
  }
```

Function with No Arguments & Return Value

→ Here the called function does not receive any data from calling function. It manages with its local data to carry out specified task.

* After the processing is complete the called function returns the computed data to calling function

* It is also a one-way data communication between calling function & called function

eg

```
# include < stdio.h >
void main ( )
    { float sum;
      float total ();
      sum = total();
      printf ("sum = %f \n", sum);
    }

float total()
    {
      float x, y;
      x = 20.0;
      y = 10.0;
    }      return (x+y);
```

Parameter Passing Mechanism

Parameter passing means the way in which the ~~p~~ arguments are given to the called function.

There are two techniques of passing parameters

(i) Call by Value

(ii) Call by reference

Call by Value

→ When the values of arguments are passed from the Calling function to the called function, the values are copied into called function.

→ If any changes are made to the value in the called function, there is no change in the original value within the calling function.

Program

```
# include <stdio.h>
main()
{
  int n1, n2, x;
  int call_by_val();
  n1 = 6;
  n2 = 9;
  printf("n1 = %d & n2 = %d\n", n1, n2);
  x = call_by_val (n1, n2);
  printf ("n1 = %d & n2 = %d \n", n1, n2);
    printf ("x = %d", x);
}

call_by_val (~~p1~~ int p1, int p2)
{
  int sum;
  sum = p1+p2;
  p1 + = 2;
  p2 + = p1;
  printf ("p1 = %d & p2 = %d\n" p1, p2);
  return (sum);
}
```

output

n1 = 6 & n2 = 9
p1 = 8 & p2 = 17
n1 = 6 & n2 = 9
x = 15

Call by Reference :

In this method, the actual values are not passed, instead their address 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, the original values get changed within calling function.

```
# include<stdio.h>
void main()
{
    int a = 10, b = 20;
    swapr(&a, &b);
    printf("\n a= %d b= %d", a, b);
}

swapr(int *x, int *y)
{ int t;
  t = *x;
  *x = *y;
```

output,
a = 20 b = 10;

Recursion :

* It is possible for a function to call itself. A function is called 'recursive' if a statement within the body of a function calls the same function
* Recursion is thus the process of defining something in terms of itself.

Program

```
# include <stdio.h>
int facto(int);
void main()
{
 int n, answer;
 printf(" Enter a number");
 scanf ("%d", &n);
 answer = facto(n);
 printf("Factorial is %d", answer);
}
```

```
jacto (int n)
{ int factorial;
  if (n==1)
  {
    return (1);
  }
  else
    {  factorial = n * jacto(n-1);
       return (factorial);
    }
}
```

<u>output:</u>

```
Enter a number 5
Factorial is 120
```

SRM INSTITUTE OF SCIENCE AND TECHNOLOG
DELHI-NCR CAMPUS

18CSS101J – Programming for Problem Solving
Unit IV

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI

UNIT IV INTRODUCTION

Passing Array Element to Function-Formal and Actual Parameters-Advantages of using Functions-Processor Directives and #define Directives-Nested Preprocessor Macro-Advantages of using Functions-Functions-Pointers and address operator-Size of Pointer Variable and Pointer-Operator-Pointer Declaration and dereferencing-pointers-Void Pointers and size of Void Pointers-Arithmetic Operations-Incrementing Pointers-Pointers-Constant Pointers-Pointers to array elements and strings-Function Pointers-Array of Function Pointers-Accessing Array of Function Pointers-Null Pointers-Pointers

INTRODUCTION

- Arrays can be passed as a parameter to a function.
- When it is being passed, the name of the array is enough as an argument instead passing it as an entire array.
- Since the name of the array itself holds the address, no need to include & in the function call.
- Since it passes address , in case of any modification occurs in called function automatically reflects back in the calling function.

Introduction

- Multidimensional arrays are also allowed to be passed as arguments to functions.
- The first dimension is omitted when a multidimensional array is used as a formal parameter in a function.

Formal and Actual Parameters One-Dimensional Array

- **Actual Parameters:**

```
int a[10];
```

Function call:

```
add(a);
```

here,

a is the base address of the array

add is the function

Formal and Actual Parameters One-Dimensional Array

- **Formal Parameters:** within the function body

```
void add(int a[])
{
....
}
```

int a[] - is a formal parameter, since it is a on dimensional

array size doesn't matter.

Formal and Actual Parameters Two-Dimensional Array

- **Actual Parameters:**

```
int a[10][10];
```

Function call:

```
add(a);
```

here,

a is the base address of the array ‘a’

b is the base address of the array ‘b’ add is the function

Formal and Actual Parameters One-Dimensional Array

- **Formal Parameters:** within the function body

```
void add(int a[][10])
{
....
}
```

int a[][10] - is a formal parameter, the first dimension is omitted only column value is taken into the account.

Passing Single element

- **Example**

int a[5]={0,1,2,3,4};

Function call:

add(a[1]); Here,

add is the function name,

a[1] is the second element of the array, value 1 is passed to the array.

Adding Constant five to the given array using functions(One-Dimensional)

Main Function

```
int main()
{
        int a[50],i,n;
        scanf("%d",&n);
        for(i=1;i<=n;i++)
                scanf("%d",&a[i]);
        add(a,n);
        for(i=1;i<=n;i++)
                printf("%d\n",a[i]);
        return 0;
}
```

Adding Constant five to the given array using functions(One-Dimensional)

- **Add Function:**

```
void add(int a[],int n)
{
int      r;
for(r=1;r<=n;r++)
{
a[r]=a[r]+5;
}
}
```

Output

- **Input:**

5

2 4 6 8 10

- **Output:**

7

9

11

13

15

Main Function

```
int main()
{
int num[2][2], i, j;  printf("Enter 4 numbers:\n");  for (i = 0; i <
2; ++i)
{
for (j = 0; j < 2; ++j)
{
scanf("%d", &num[i][j]);
}
}
// passing multi-dimensional array to displayNumbers function
displayNumbers(num);
return 0;
}
```

Display Numbers Function

```
void displayNumbers(int num[2][2])
{
// Instead of the above line,
// void displayNumbers(int num[][2]) is also valid  int i, j;
printf("Displaying:\n");
for (i = 0; i < 2; ++i)
{
for (j = 0; j < 2; ++j)
{
printf("%d\n", num[i][j]);
}
}
}
```

The advantages of using functions are:

- Divide and conquer
- Manageable program development
- Software reusability
- Use existing functions as building blocks for new programs
- Abstraction - hide internal details (library functions)
- Avoid code repetition

Preprocessor Directives MACROS

- The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation.
- A macro is a segment of code which is replaced by the value of macro.
- Macro is defined by #define directive.
- Preprocessing directives are lines in your program that start with #.
- The # is followed by an identifier that is the directive name.
- For example, #define is the directive that defines a macro.
- Whitespace is also allowed before and after the #.

List of preprocessor directives

#include
#define
#undef
#ifdef
#ifndef
#if
#else
#elif
#endif
#error
#pragma

#include

- The #include preprocessor directive is used to paste code of given file into current file.
- It is used include system-defined and user-defined header files.
- If included file is not found, compiler renders error. It has three variants:

#include <file>

- This variant is used for system header files.
- It searches for a file named file in a list of directories specified by us, then in a standard list of system directories.

#include "file"

- This variant is used for header files of your own program.
- It searches for a file named file first in the current directory, then in the same directories used for system header files.
- The current directory is the directory of the current input file.

#include anything else

- This variant is called a computed #include.
- Any #include directive whose argument does not fit the above two forms is a computed include.

Macro's (#define)

#define token value There are two types of macros:

1.Object-like Macros

2.Function-like Macros

<u>Object-like Macros</u>

- The object-like macro is an identifier that is replaced by value.
- It is widely used to represent numeric constants.

For example:

```
#include <stdio.h>  #define PI 3.1415  main()
{
printf("%f",PI);
}
```

Output: 3.14000

Function-like Macros

- The function-like macro looks like function call.
- For example:#define MIN(a,b) ((a)<(b)?(a):(b))
- Here, MIN is the macro name. #include <stdio.h>

```
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output:

Minimum between 10 and 20 is: 10

#undef

- To undefine a macro means to cancel its definition.

This is done with the #undef directive.

Syntax:

```
#undef token
```

define and undefine example

```
#include <stdio.h>
#define PI 3.1415
#undef PI
main()
{       printf("%f",PI); }
```

Output

Compile Time Error: 'PI' undeclared

#ifdef

The #ifdef preprocessor directive checks if macro is defined by #define.
If yes, it executes the code.

Syntax:

#ifdef MACRO //code #endif

#ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code.

Syntax:

#ifndef MACRO //code #endif

#if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code.

Syntax:

#if expression //code #endif

#else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

#if expression //if code #else //else code #endif

Syntax with #elif

#if expression //if code #elif expression //elif code #else //else code #endif

Example

```
#include <stdio.h>  #include <conio.h>
#define NUMBER 1  void main()
{
#if NUMBER==0
printf("Value of Number is: %d",NUMBER);
#else print("Value of Number is non-zero");
#endif getch();
}
```

Output

Value of Number is non-zero

#error

- The #error preprocessor directive indicates error.
- The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

```
#include<stdio.h>
#ifndef _____MATH_H
#error First include then compile
#else
void main(){
float a; a=sqrt(7);  printf("%f",a);
}
#endif
```

#pragma

- The #pragma preprocessor directive is used to provide additional information to the compiler.
- The #pragma directive is used by the compiler to offer machine or operating-system feature.
- Different compilers can provide different usage of #pragma directive.

Syntax:

#pragma token

Example

```
#include<stdio.h>
#include<conio.h>  void
func() ;  #pragma
startup func  #pragma
exit func  void main()
{
printf("\nI am in main");
getch(); }
void func(){
printf("\nI am in func");
getch();
}
```

Output

I am in func
I am in main I am in func

Advantages of Macros

- Time efficiency.
- Not need to pass arguments like function.
- It's preprocessed.
- Easier to Read

Disadvantages of Macros

- Very hard to debug in large code.
- Take more memory compare to function

POINTERS

Definition

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

POINTERS

- Syntax

  Datatype *pointervariable;

- Syntax Example

```
1. int    *ip;    /*  pointer to an integer*/
2. double *dp;    /*  pointer to a double*/
3. float  *fp;    /*  pointer to a float*/
4. char   *ch     /*  pointer to a character*/
```

POINTERS

- Example

```
int  var = 20;          /* actual variable declaration */

int  *ip;               /* pointer variable declaration */

ip = &var;              /* store address of var in pointer
                           variable*/
```

POINTERS

Reference operator (&) and Dereference operator (*)

1.& is called reference operator. It gives the address of a variable.

2.* is called dereference operator. It gives the value from the address

Pointer to Pointer

Double (**) is used to denote the double pointer. Double Pointer Stores the address of the Pointer Variable. Conceptually we can have Triple ….. n pointers.

Example 1

```
int main()
{
int num = 45 , *ptr ,
**ptr2ptr ;  ptr          = &num;
          //3000
ptr2ptr = &ptr;          //4000
printf("%d",**ptr2ptr);
return(0);
}
```

Output 45

Pointer to Constant Objects

These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.

```
const datatype *pointername;
(or)  datatype const *pointername;
```

Example

The pointer variable is declared as a const. We can change address of such pointer so that it will point to new memory location, but pointer to such object cannot be modified (*ptr).

Example

Turbo C++ IDE

File Edit Search Run Compile Debug Project Options Window Help

CPOINTER.C 1

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int num[2]={20,70};
        const int *ptr;
        clrscr();
        ptr=&num[0];
        ptr++;
        num[1]=20;
        *ptr=30;
        printf("%d\n%d",*ptr,ptr);
        getch();
}
```

11:14

Message

Compiling CPOINTER.C:

•Error CPOINTER.C 11: Cannot modify a const object

F1 Help Space View source Edit source F10 Menu

Constant Pointers

Constant pointers are the one which cannot change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). If we try to point the pointer to some other variable, then it is not possible.

```
int* const ptr=&variable;  (or)
int *const ptr=&variable // ptr is a constant pointer to int
```

Null pointer

- NULL Pointer is a pointer which is pointing to nothing.
- Pointer which is initialized with NULL value is considered as NULL pointer.

```
datatype *pointer_variable=0;  datatype

*pointer_variable=NULL;
```

Example

Turbo C++ IDE

File Edit Search Run Compile Debug Project Options Window Help

CONPOI.C

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

        int *ptr=NULL;
        clrscr();
        printf("%d",ptr);
        getch();
}
```

5:16

F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

Pointer Arithmetic

- C allows you to perform some arithmetic operations on pointers.
- **Incrementing a pointer**

Incrementing a pointer is one which increases the number of bytes of its data type.

```
int *ptr;
int a[]={1,2,3};
ptr=&a;
ptr++;
ptr=&a;
ptr=ptr+1;
```

Pointer Arithmetic

- **Decrementing a Pointer**

Decrementing a pointer is one which decreases the number of bytes of its data type.

Using Unary Operator

```
int *ptr;
int a[]={1,2,3};
ptr=&a;
ptr--;
```

Limitations of Pointer Arithmetic

- Addition of 2 pointers is not allowed
- Addition of a pointer and an integer is commutative ptr+5**ó** 5+ptr
- Subtraction of 2 pointers is applicable.
- subtraction of a pointer and an integer is not commutative ptr-**5≠ 5**-ptr.
- Only integers can be added to pointer. It is not valid to add a float or double value to a pointer.
- A pointer variable cannot be assigned a non address value except zero.
- Multiplication and division Operators cannot be applied on pointers.
- Bitwise operators cannot be applied on pointers.
- A pointer and an integer can be subtracted.
- A pointer and an integer can be added.

Void pointer

1. Void pointer is a generic pointer and can point to any type of object.The type of object can be char, int, float or any other type.

- **Example**

```
File  Edit  Search  Run
#include<conio.h>
void main()
{
        int a=10;
        void *ptr=&a;
        clrscr();
        printf("%u",ptr);
        getch();
}
```

Turbo C++ IDE

```
65524
```

2.A pointer to any type of object can be assigned to a void pointer.

Turbo C++ IDE

File Edit Search Run Compile Debug Project Options Window Help

CONPOI.C

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a=10;
        int *iptr=&a;
        void *vptr=iptr;
        clrscr();
        printf("int* is implicitely converted to void* %u\t%u",vptr,iptr);
        getch();
}
```

10:37

F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

OUTPUT

Turbo C++ IDE

```
int* is implicitely converted to void* 65524    65524
```

3. A void pointer cannot be dereferenced

Turbo C++ IDE

File Edit Search Run Compile Debug Project Options Window Help

CONPOI.C 1

```
#include<conio.h>
void main()
{

        int a=10;
        void *ptr=&a;
        clrscr();
        printf("%d",*ptr);
        getch();
}
```

Message 2=[↑]

```
Compiling CONPOI.C:
•Error CONPOI.C 9: Not an allowed type
Warning CONPOI.C 11: 'ptr' is assigned a value that is never used
```

F1 Help Space View source ↵ Edit source F10 Menu

Relational operations

- A pointer can be compared with a pointer of same type or zero.
- Various relational operators are ==.,!=,<,<=,>,>=
- Ex: float a=1.0,b=2.0,*fptr1,*fptr2;
- fptr1=&a;fptr2=&b;
- int result;

result=fptr1!=fptr2;

Example

Turbo C++ IDE

File Edit Search Run Compile Debug Project Options Window Help

CONPOI.C

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

        float *ptr1,*ptr2,a=1.0,b=2.0;
        int result;
        ptr1=&a;ptr2=&b;
        result=ptr1!=ptr2;
        clrscr();
        printf("%d",result);
        getch();
}
```

9:21

F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

Pointers and Arrays

- Pointers and arrays are closely related , An array variable is actually just a pointer to the first element in the array.
- Accessing array elements using pointers is efficient way than using array notation.
- When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array.
- Base address i.e address of the first element of the array is also allocated by the compiler.

Pointers and Arrays Cont…

Suppose we declare an array **arr**,

int arr[5]={ 1, 2, 3, 4, 5 };Assuming that the base address of **arr** is 1000 and each integer requires two bytes, the five elements will be stored as follows

Pointers and Arrays Cont…

- Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**.
- Therefore **arr** is containing the address of **arr[0]** i.e 1000. In short, arr has two purpose- it is the name of an array and it acts as a pointer pointing towards the first element in the array.

Pointers and Arrays Cont…

int *p; p = arr; or

p = &arr[0];

Now we can access every element of array **arr** using **p++** to move from one element to another.

Pointers and Arrays Cont…

- a[0] is the same as *a
- a[1] is the same as *(a + 1)
- a[2] is the same as *(a + 2)
- If pa points to a particular element of an array, (pa + 1) always points to the next element, (pa + i) points i elements after pa and (pa - i) points i elements before.

Example 1

```
#include<stdio.h>  void
main()
{
int a[10],i,n;  printf("Enter
n");  scanf("%d",&n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n;i++)
printf("a[%d]=%d\n",i,*(a+i)
);
}
```

Output

```
Enter n
5
1 2 3 4
5
a[0]=1
a[1]=2
a[2]=3
a[3]=4
a[4]=5
```

Pointer to Multidimensional Array

- A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array.
- As we know now, name of the array gives its base address. In a[i][j], **a** will give the base address of this array, even a+0+0 will also give the base address, that is the address of **a[0][0]** element.
- Here is the generalized form for using pointer with multidimensional arrays.
- ***(*(a + i) + j)** is same as a[i][j].

Example 2

```
#include <stdio.h>  #define ROWS 4
#define COLS 3  int main ()
{
int i,j;
// declare 4x3 array
int matrix[ROWS][COLS] = {{1, 2, 3},
                          {4, 5, 6},
                          {7, 8, 9},
                          {10, 11, 12}};
for (i = 0; i < ROWS; i++)
{
for (j = 0; j < COLS; j++)
{
        printf("%d\t", matrix[i][j]);
}
printf("\n");
}
return 0;
}
```

Output

1 2 3

4 5 6

7 8 9

10 11 12

matrix[0][0] = *(*(matrix))

matrix[i][j] = *((*(matrix)) + (i * COLS + j)) matrix[i][j] = *(*(matrix + i) + j)

matrix[i][j] = *(matrix[i] + j) matrix[i][j] = (*(matrix + i))[j]

&matrix[i][j] = ((*(matrix)) + (i * COLS + j))

Example 3

```
void array_of_arrays_ver(int
arr[][COLS])
{
int i, j;
for (i = 0; i < ROWS; i++)
{
for (j = 0; j < COLS; j++)
{
printf("%d\t", arr[i][j]);
}
printf("\n");
}
}
void ptr_to_array_ver(int (*arr)[COLS])
{
int i, j;
for (i = 0; i < ROWS; i++)
{
for (j = 0; j < COLS; j++)
{
printf("%d\t", (*arr)[j]);
}
arr++;  printf("\n");
}
}
```

Output

C:\Users\prakash\Desktop\dektopfile\unit5pdd\pro3.exe

```
Printing Array Elements by Array of Arrays Version Function:
1       2       3
4       5       6
7       8       9
10      11      12
Printing Array Elements by Pointer to Array Version Function:
1       2       3
4       5       6
7       8       9
10      11      12

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

Double Pointer

```
#include <stdio.h>  #define ROWS 4
#define COLS 3
int main ()
{
// matrix of 4 rows and 3 columns
int matrix[ROWS][COLS] = {{1, 2, 3},
{4, 5, 6},
{7, 8, 9},
{10, 11, 12}};
int** pmat = (int **)matrix;
printf("&matrix[0][0] = %u\n", &matrix[0][0]);
printf("&pmat[0][0] = %u\n", &pmat[0][0]);
return 0;
}
```

Output

&matrix[0][0] = 2675498000

&pmat[0][0] = 1

Passing an array to a function

- Single element of an array can be passed in similar manner as passing variable to a function
- **C program to pass a single element of an array to function**

```
#include <stdio.h>  void display(int age)
{
printf("%d", age);
}

int main()
{
int ageArray[] = { 2, 3, 4 };
display(ageArray[2]); //Passing array element ageArray[2]
    only.
        return 0;
    }
```

OUTPUT

4

Passing an entire one-dimensional array to a function

```
#include<stdio.h>
int total(int[],int);
void main()
{
int a[10],n,sum,i;
printf("Enter n\n");
scanf("%d",&n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
sum=total(a,n);
printf("Sum=%d",sum);
}
int total(int a[],int n)
{
int sum=0,i;
for(i=0;i<n;i++)
sum=sum+a[i];  return
sum;
}
```

Output

Enter n

5

1

2

3

4

5

Sum=15

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example 4

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
int num[2][2], i, j;  printf("Enter 4
numbers:\n");  for (i = 0; i < 2; ++i)
for (j = 0; j < 2; ++j)
scanf("%d", &num[i][j]);
// passing multi-dimensional array to displayNumbers function
displayNumbers(num);  return 0;
}
```

```
void displayNumbers(int num[2][2])
{
// Instead of the above line,
// void displayNumbers(int num[][2]) is also valid
int i, j;  printf("Displaying:\n");  for
(i = 0; i < 2; ++i)
for (j = 0; j < 2; ++j)
printf("%d\n", num[i][j]);
}
```

Output

Enter 4
numbers:
1
2
3
4
Displaying: 1
2
3
4

Pointers and Strings

- **Strings as arrays:**In C, the abstract idea of a string is implemented with just an array of characters. For example, here is a string:
- char label[] = "Single";What this array looks like in memory is the following:

S	i	n	g	l	e	\0

Pointers and Strings cont…

- where the beginning of the array is at some location in computer memory, for example, location 1000.
- A character array can have more characters than the

abstract string held in it, as below:

- char label[10] = "Single"; giving an looks like:

- we can access each character in the array using subscript notation, as in:

```
printf("Third char is: %c\n", label[2]);
```

- which prints out the third character, **n**.

Disadvantage Of Creating Strings Using The Character Array

- A disadvantage of creating strings using the character array *syntax* is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

```
char label[] = "Single"; /* 7 characters */  char label[10] = "Single";
```

- Thus, you must specify the maximum number of characters you will ever need to store in an array.
- This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

String as Pointers

- Another way of accessing a *contiguous* chunk of memory, instead of with an array, is with a *pointer*.
- However, pointers only hold an address, they cannot hold all the characters in a character array.
- This means that when we use a char * to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

```
char label[] = "Single";
char label2[10] = "Married";  char *labelPtr;
labelPtr = label;
```

Example1

```
#include <stdio.h>  int main()
{
char *sample = "From whence cometh my
help?n";  while(putchar(*sample++))
;
return(0);
}
```

Output:

From whence cometh my help?n

Passing Strings

Below is the definition of a function that prints a label and a call to that function:

```
void PrintLabel(char the_label[])
{
printf("Label: %s\n", the_label);
                                    }
...
int main(void)
{
char label[] = "Single";
...
PrintLabel(label);
...
}
```

Passing Strings Cont...

- Since label is a character array, and the function PrintLabel() expects a character array, the above makes sense.
- However, if we have a pointer to the character array label, as in:

```
char *labelPtr = label;
```

then we can also pass the pointer to the function, as in:

- PrintLabel(labelPtr);

Array of Pointers

- Pointers are very helpful in handling character array with rows of varying length.

char *name[3]={ "Adam", "chris", "Deniel" }; *//Now see same array without using pointer* char name[3][20]= { "Adam", "chris", "Deniel" };

Example1

```
#include <stdio.h>  #define SIZE 3
int main()
{
char president[SIZE][8] = {
"Clinton",
"Bush",  "Obama"
};
int x,index;  for(x=0;x<SIZE;x++)
{
index = 0;
while(president[x][index] != '\0')
{
putchar(president[x][index]);
index++;
}
putchar('n');
}
return(0);
}
```

Output:

ClintonnBushnObaman

Function Pointers

- In C, like normal data pointers (int *, char *, etc), we can have pointers to functions.
- Initialization

return_type function_pointer(argu)=&function_name

```
void (*fun_ptr)(int) = &fun;
```

- Function Definition

```
void fun(int a)
{
printf("Value of a is %d\n", a);
}
```

Example1

```
#include<stdio.h>  void
fun(int a)
{
printf("a=%d\n",a);
}
void main()
{
void (*fun1)(int)=&fun;
(*fun1)(15);
}
```

Output:

a=15

Function Pointers Cont…

If we remove bracket, then the expression

"void (*fun_ptr)(int)"

becomes

"void *fun_ptr(int)"

which is declaration of a function that returns void pointer.

Interesting facts

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function's name can also be used to get functions' address.

Passing Function Pointer as an argument

```
#include<stdio.h>
void
fun1(void(*test)())
{
int a=20;  test(a);
}
void test(int a)
{
printf("a=%d\n",a);
}
void main()
{
fun1(&test);
}
```

Output:

a=20

Array of Function Pointer

```
#include<stdio.h>  void
add(int a,int b)
{
printf("add=%d\n",a+b);
}
void sub(int a,int b)
{
printf("sub=%d\n",a-b);
}
void mul(int a,int b)
{
printf("mul=%d\n",a*b);
}
```

```
void main()
{
void
(*fun[])(int,int)={add,sub,mul}
;
int ch;
int a,b;
printf("Enter a and b\n");
scanf("%d%d",&a,&b);
printf("Enter the
operation\n");
scanf("%d",&ch);  if(ch>2)
printf("Wrong Input\n");  else
(*fun[ch])(a,b);
}
```

Output

Enter a and b 4

5

Enter the

operation 2

mul=20

Structures created and accessed using pointers

- Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name
{
member1;
member2;
. .
};
int main()
{
struct name *ptr;
}
```

Here, the pointer variable of type struct name is created.

Accessing structure's member through pointer

A structure's member can be accessed through pointer in two ways:

1. Referencing pointer to another address to access memory

2. Using dynamic memory allocation

Referencing pointer to another address to access the memory

Consider an example to access structure's member through pointer.

```
#include <stdio.h>
typedef struct person
{           int age;  float weight;
};
int main()
{
struct person *personPtr, person1;
personPtr = &person1;          // Referencing pointer to memory address of person1
printf("Enter integer: ");
scanf("%d",&(*personPtr).age);  printf("Enter number: ");
scanf("%f",&(*personPtr).weight);  printf("Displaying:\n");
printf("age=%d\nweight=%f",(*personPtr).age,(*personPtr).weight);  return 0;
}
```

Output

```
Enter integer: 3
Enter number: 6
Displaying:
age=3
weight=6.00000
0
```

->

- **Using -> operator to access structure pointer member.**
- Structure pointer member can also be accessed using -> operator.
- (*personPtr).age is same as personPtr->age
- (*personPtr).weight is same as personPtr->weight

Accessing structure member through pointer using dynamic memory allocation

- To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.
- **Syntax to use malloc()**

ptr = (cast-type*) malloc(byte-size)

Example1

```
#include <stdio.h>  #include <stdlib.h>  struct
person {
int age;
float weight;  char name[30];
};
int main()
{
struct person *ptr;  int i, num;
printf("Enter number of persons: ");
scanf("%d", &num);
ptr = (struct person*) malloc(num *
sizeof(struct person));
// Above statement allocates the memory for n
structures with pointer personPtr pointing to
base address */
```

```
for(i = 0; i < num; ++i)
{
printf("Enter name, age and weight of the
person respectively:\n");
scanf("%s%d%f", &(ptr+i)->name,
&(ptr+i)->age, &(ptr+i)->weight);
}

printf("Displaying Infromation:\n");  for(i
= 0; i < num; ++i)
printf("%s\t%d\t%.2f\n", (ptr+i)->name,
(ptr+i)->age, (ptr+i)->weight);
return 0;
}
```

Output

```
Enter number of persons: 2
Enter name, age and weight of the person
respectively:
aaa  12
45
Enter name, age and weight of the person
respectively:  bbb
87
65
Displaying Infromation:
aaa     12      45.00
bbb     87      65.00
```

Self-Referential Structure

- A self referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

```
struct struct_name
{
datatype datatypename;  struct_name *
  pointer_name;
};
```

Self-Referential Structure Cont…

- A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type.
- For example, a linked list is supposed to be a self-referential data structure.The next node of a node is being pointed, which is of the same struct type. For example,

```
typedef struct listnode {  void *data;
struct listnode *next;
} linked_list;
```

In the above example, the listnode is a self-referential structure

– because the *next is of the type struct listnode.

Self-Referential Structure Cont…

```
typedef struct
   listnode {  void
   *data;
struct listnode *next;
} linked_list;
```

THANK YOU

Array:

→ An array can be defined as an ordered list of homogeneous data elements.

→ These elements may be of type int, float, char or double.

→ All the elements in array are stored in consecutive memory location

→ An array is described by a single name and each individual data item in the array is referenced by a subscript (or index) enclosed in a pair of square bracket.

→ This subscript indicates the position of an individual data item in an array

→ In C language, the subscript value ranges from 0 to one less than the maximum size. If the size of an array is 10, Then, the first subscript is 0, the second subscript is 1 & so on. The last subscript is 9. In general, the ith element has subscript $(i-1)$.

eg int arr[10];

arr[0], arr[1], arr[2] - - - - - arr[9].

Memory Representation of Array

intarr[10];

		arr[2]		arr[4]		arr[6]		arr[8]	
41	56	20	01	6	46	98	108	999	1024
arr[0]	arr[1]		arr[3]		arr[5]		arr[7]		arr[9]

Declaration of Array

It is a linear list of fixed number of data items of same type. All these data items are accessed using the same name using a single subscript.

Syntax

datatype arrayname [size];

Where

datatype → type of data stored in ~~array~~ array

arrayname → name of array

size → maximum no. of elements that can be stored in an array.

Total Memory size of Array

Total size = size * (size of data type)

where:

size → number of elements in one dimensional array

size of datatype → memory requirement of data type.

Qe Program to accept n integers & store them in an array called num. Also print them

```
# include < stdio.h >
void main()
{
    int n, i, num[20];
    printf(" Enter the size of an array\n");
    scanf("%d", &n);
    for (i=0; i<n; i++)
    {
        scanf("%d", &num[i]);
    }
```

```
printf ("Array is \n");
    for (i=0; i<n; i++)
      {
        printf ("num [%d] = %d \n", i, num[i]);
      }
    }
```

Program to Read marks of 5 Subjects in array:

```
# include <stdio.h>
void main( )
    {
      int i;
      float marks [5];

    for (i=0; i<=4; i++)
      {
        scanf ("%f", & marks[i]);
      }
```

Program (Array)

Q.1 Program to arrange elements of array in increasing order.

```
# include <stdio.h>
Void main ()
{
    float array [20];
      int i, j, n, temp;
    printf (" Enter the size of an array\n");
    scanf ("%d", &n);
    printf ("Enter elements \n");
      for (i=0; i<n; i++)
          scanf ("%f", & array[i]);
    printf (" Input array is \n");
          for (i=0; i<n ; i++)
              printf ("%f\n", array [i]);
    for (i=0; i<n-1; i++)
      {
          for (j= i+1 ; j<n; j++)
        {
          if (array [i] >= array [j])
```

```
{
        temp = array[i];
        array[i] = array[j];
        array[j]= temp;
    }
    }
}

printf ("Sorted array is ... \n");
    for (i=0; i<n; i++)
        printf ("%f \n", array[i]);
    }
```

Initializing a One dimensional Array

→ Initializing means assigning some values to the variable that undergoes processing.

→ Like other variables, elements of array can also be initialized. All initialized values should be constants.

→ It is possible to declare all the variables of array during its declaration

Syntax

datatype array-name [size] = {element1, element2 ... elementn};

where

datatype → data type of elements stored

array-name → name of array

size → maximum number of elements

element1, element2 ... → initial value of data elements

eg

```
int evennum[4] = {2, 4, 6, 8};
```

Note: If no value is explicitly assigned to any subscript of array it takes zero as initialised value.

→ It is also possible to declare arrays with their initial values, without writing any value within brackets.

eg int age[ ] = { 16, 26, 18, 24, 23, 52 }

here

age [0] = 16
age [1] = 26
age [2] = 18
age [3] = 24
age [4] = 23
age [5] = 52

Program

```
# include <stdio.h>
main ()
{
    int marks [ ] = { 45, 65, 90, 87, 74 };
    int i;
    printf (" Elements of array are\n");
    for (i=0; i<5; i++)
        printf ("marks [%d] = %d \n", i, marks[i]);
}
```

Q6 Program to accept two integer array & find sum of corrosponding elements of these array.

```
# include <stdio.h>
void main()
{
 int arr1[10], arr2[10], sum[10];
 int i, n;
 printf("Enter size of arrays\n");
 scanf("%d", &n);
 printf("Enter elements of array1\n");
 for(i=0; i<n; i++)
    scanf("%d" & arr1[i]);
 printf("Enter elements of array2\n");
    for (i=0; i<n; i++)
       scanf("%d", & arr2[i]);
 for (i=0; i<n; i++)
    sum[i] = arr1[i] + arr2[i];
 printf("sum of elements of array1 & array2\n");
    for(i=0; i<n; i++)
    printf("%d\n", sum[i]);
```

Q. Program to print largest element of an array size 10;

```
# include <stdio.h>
void main()
{
    int x[10], large, i;
    printf ("Enter elements of array\n");
    ~~scanf(~~
    for (i=0; i<10; i++)
        scanf ("%d", &x[i]);
        large = x[0];
        for (i=1; i<10; i++)
        {
            if(x[i] > large)
                large = x[i];
        }
        printf ("largest element is %d", large);
}
```

Q6 Program to search desired element in n sized array

```
# include < stdio. h >
void main ()
{
    int n, i, found=0, listn [20], key_element;
  printf (" Enter number of elements in array\n");
  scanf ("%d", &n);
   printf (" Enter %d numbers \n", n);
    for (i=0; i<n ; i++)
        scanf ("%d", & listn[i]);
    printf ("Enter key element to search\n");
     scanf ("%d", & key_element);
    for (i=0; i<n; i++)
     {
       if (listn [i] == key_element)
          found =1;
     }
     if (found == 1)
        printf (" Element is found");
    else
        printf (" Element is not found");
}
```

Program to compute sum of even & odd numbers stored in an array.

```
#include <stdio.h>
void main()
{
    int num[20];
    int i, esum=0, odsum=0,n;
    printf ("Enter size of array \n")
    scanf ("%d", &n);
    printf (" Enter the elements of array \n");
        for (i=0; i<n; i++)
            scanf ("%d", & num[i]);
        for (i=0; i<n; i++)
        {
            if(num[i] % 2 == 0)
                esum + = num[i];
            else
                odsum + = num[i];
        }
    printf ("Sum of even numbers is %d", esum);
    printf ("Sum of odd numbers is %d", odsum);
}
```

Multidimensional Array:

If the number of subscript is more than one then such array is called multidimensional array.

eg - 2D (two dimensional)
3D (three dimensional)

Two Dimensional Array

2D array is an ordered table of homogeneous elements. It is generally, reffred to as matrix of some rows & some columns.

Syntax

datatype array-name [rows] [columns];

where

datatype → types of data stored in array
array_name → name of array
row → number of elements to be processed under subscript 1
Column → number of elements to be processed under subscript 2

eg. int marks [5] [3];

In this example array has 5 rows & 3 columns.

Structure of array is like:

marks [0] [0] =	marks [0] [1] =	marks [0] [2] =
marks [1] [0] =	marks [1] [1] =	marks [1] [2] =
marks [2] [0] =	marks [2] [1] =	marks [2] [2] =
marks [3] [0] =	marks [3] [1] =	marks [3] [2] =
marks [4] [0] =	marks [4] [1] =	marks [4] [2] =

Initialization of 2-D Array

The elements of two dimensional can also be initialized at the time of declaration

Syntax

datatype arrayname [size1] [size2] = {e1, e2 - - - en};

where → datatype → Type of data elements in array

arrayname → name of array

e1, e2 - - en → initial values to be assigned to n elements of array

eg int matrix [3] [3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;

matrix [0] [0] = 1 matrix [0] [1] = 2 matrix [0] [2] = 3

matrix [1] [0] = 4 matrix [1] [1] = 5 matrix [1] [2] = 6

matrix [2] [0] = 7 matrix [2] [1] = 8 matrix [2] [2] = 9

Q. Program to Read & Print elements of a Matrix.

```
# include <stdio.h>
  Void main( )
    {
      int n, i, j, row, col;
      int mat [10] [10] ;
     printf (" Enter order of matrix \n");
      scanf ("%d %d", &row, & col);
      printf (" Enter elements of array (matrix) \n");
      for (i=0; i< row ; i++)
        {
          for (j=0; j<col; j++)
            {
              scanf ("%d", & mat[i][j]);
            }
        }
```

```
printf("Matrix is \n");
 for (i=0; i<row; i++)
   {
     for (j=0; j<col; j++)
       {
         printf("%d", mat[i][j]);
          printf("\t");
       }
     printf("\n");
   }
}
```

<u>Output</u>

```
Enter order of matrix
  3   3

Enter elements of array (matrix)
  1
  2
  3
  4
  5
  6
  7
  8
  9

Matrix is
  1   2   3
  4   5   6
  7   8   9
```

~~Matrix is~~

Q. Program to accept two matrix of same order & find the sum of these matrix & print sum

```
# include <stdio.h>
void main()
{
    int n, i, j, row, col;
    int mat1 [10] [10], mat2 [10] [10], matsum [10] [10];
    printf (" Enter Rows & Columns\n");
    scanf ("%d %d", & row, & col);
    printf (" Enter elements of matrix 1\n");
    for (i=0; i< row ; i++)
    {
        for (j=0; j< col; j++)
        {
            scanf ("%d", & mat1[i] [j]);
        }
    }
    printf (" Enter elements of matrix 2\n");
    for (i=0; i< row; i++)
    {
        for (j=0; j< col; j++)
        {
            scanf ("%d", & mat2 [i] [j]);
        }
    }
```

```
for (i=0; i< row ; i++)
  {
    for (j=0; j<col; j++)
      {
        matsum[i][j] = mat1[i][j] + mat2[i][j];
      }
  }

printf (" The sum matrix is..

for (i=0; i< row; i++)
  {
    for (j=0; j< col ; j++)
      { printf ("%d", matsum[i][j]);
      }

      printf ("\n");
  }

}
```

Program to find product of two matrices & print the product matrix.

```
# include <stdio.h>
  void main()
   {
     int i, j, k, row1, col1, row2, col2;
    int matA [10] [10], matB [10] [10], prodmat[10][10];
    printf ("Enter order of matrix 1 \n");
     scanf ("%d %d", & row1, & col1);
     printf ("%d %d", &
      printf ("Enter order of matrix 2 \n");
      scanf ("%d %d", & row2, & col2);
    if (col1 == row2)
     {
       printf ("Enter the elements of Matrix 1 \n");
      for (i=0; i<row1; i++)
        {
          for(j=0; j<col1; j++)
           {
             scanf ("%d", & matA [i] [j]);
           }
        }
     printf (" Enter elements of Matrix 2 \n");
```

```
for (i=0; i<row2; i++)
{
    for (j=0; j<col2; j++)
    { scanf ("%d", &matB[i][j]);
    }
}

for (i=0; i<row1; i++)
{
    for (j=0; j<col2; j++)
    {
        prodmat[i][j]=0 ;
        for (k=0; k<row2; k++)
        {
        prodmat[i][j]= prodmat[i][j]+ (matA[i][k] *
                                        matB[k][j]);
        }
    }
}
printf (" Product matrix is ... \n");
    for(i=0; i<row1; i++)
    { for (j=0; j<col2; j++)
      {
        printf("%d", prodmat[i][j]);
      }
        printf ("\n");
    }
}
else
{ printf (" Multiplication is Not Possible \n");
}
```

Some functions of headerfile <math.h>

sqrt : To find square root of number

Syntax

```
sqrt (number);
```

eg

```
# include <stdio.h>
# include <math.h>
void main ( )
{
    float param, result;
    param = 1024;
    result = sqrt (param);
printf (" Square root of %f is %f", param, result);
}
```

Output :

Square root of 1024.000000 is 32.000000

pow: To find power of one number in respect of another number.

eg: $2^3 = 8$;

syntax

```
pow (2, 3);
```

eg

```
# include <stdio.h>
# include <math.h>
void main()
{
    int a, b, result;
      a = 2;
      b = 3;
    result = pow(a, b);

  printf ("The result of %d raise to power %d is %d",
                                  a, b, result);

}
```

String :

A string in C is defined as the sequence of characters terminated by the special character '\0'.

→ The special character '\0' is called null character and it is to indicate the end of string.

→ String is also called as array of characters & an array of characters is declared and stored in memory as follows:

~~char S[] = "welcome"~~ char S[] = {"welcome"};

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]
w	e	l	c	o	m	e	\0

Initialization of Array of char Type

* char str1[5] = {"'a', 's', 'd', 'f', '\0'"};

→ here str1 is of size five, It can hold maximum five characters.

→ The initializer-list consists of comma seprated Character Constants

→ The null character is explicitly listed in this representation.

*

Char str2[5] = {"asdf"};

→ Here the string str2 is declared of length 5, It can hold maximum 5 characters including 'null' character.

→ The initializer-list consists of a string constant

→ In this representation the '\0' character is automatically appended at the end of string.

<u>Note:</u> In either of representation, the size of character array can be skipped.

eg: Char S1[] = {"abcd"};

Char S2[] = {'a', 'b', 'c', '\0'};

String I/O (Input, Output)

⇒ Using scanf & printf

* To accept string into string variable 's'.

```
scanf ("%s", s);
```

This syntax will accept string into 's' upto white space character (Blank space, New line character, Tab space).

The '\0' null character will be automatically append to the string.

Note: Symbol "&" is not attached before the name of string variable because variable name itself gives the address of string.

* To print string stored in variable 's'

```
printf ("%s", s);
```

This syntax displays the string contained in variable 's'.

Program to illustrate string initialization & string I/O using scanf & printf.

```
# include<stdio.h>
# include <conio.h>
void main()
{
    clrscr();
    char s1[10] = {"welcome"};
    char s2[10] = {"'a', 'd', 'j', 'g', '\0' "}
    char s3[] = {"welcome"};
    char s4[10];
    printf ("s1 = %s \n", s1);
    printf ("s2 = %s \n", s2);
    printf ("s3 = %s \n", s3);
    printf ("Enter string into s4 \n");
    scanf ("%s", s4);
    printf ("s4 = %s \n", s4);
    getch();
}
```

getchar() & putchar() functions

* getchar() is used to read a character through standard input, keyboard. The syntax of using getchar()

```
c = getchar();
```

where 'c' is a variable of char type, the value entered through keyboard is assigned to the variable 'c'

* putchar() is used to display a character on standard output screen. The syntax is

```
putchar(c);
```

where 'c' represents a variable of type char, it displays the character stored in 'c' on screen

Note: Both getchar() & putchar() can be used repeatedly to accept line of text and display line of text respectively.

Q2 Program to accept line of text & display it using getchar() & putchar().

```
# include <stdio.h>
# include <conio.h>
void main()
{
    Char text[100], c;
    int i;
    printf ("Enter a line of text\n");
    i=0;
    c=getchar();
    while (c!='\n')
        {
        text[i] = c;
        c= getchar();
        i++;
        }
    text[i]='\0';
printf("The line you entered is \n");
for (i=0; text[i]!='\0'; i++)
    putchar (text[i]);
} getch();
```

gets( ) & puts( )

⇒ The purpose of gets( ) is to accept string upto a new line character into string variable.

It automatically appends null character '\0' to the end of string.

Syntax

```
Char str[100];
gets(str);
```

⇒ The purpose of puts( ) is to display string contained in string variable.

It automatically adds newline character '\n' to end of string, as a result of which the cursor is moved down by one line after the string is displayed.

Syntax

```
puts(str);
```

Q. Program to illustrate gets() & puts()

```
# include<stdio.h>
# include <conio.h>
void main()
{ clrscr();
  Char str[100];
  printf ("Enter a line of text\n");
    gets (str).
  printf ("The line you entered is\n");
    puts (str);
  getch();
}
```

Output:

```
Enter a line of text
I am B.Tech Student

The line you entered is
I am B.Tech Student
```

ASCII

Full form of ASCII is American Standard Code for Information Interchange.

Each & every Alphabet (both in uppercase & Lowercase) is assigned a numeric value which indicates the value of that alphabet in Language.

Such as

A	B	C	D	E	.....	Y	Z
65	66	67	68	69	.....	89	90

Also

a	b	c	d	e	.....	y	z
97	98	99	100	101	.....	121	122

The ASCII value of the alphabets can be used directly in program written in C language.

⇒ Character variables & character constant can be included as a part of arithmetic & relational expressions.

⇒ The ASCII value of characters are evaluated that are included in expressions.

eg: 5 + 'a' = 5 + 97 = 102 {ASCII value of a is 97}

'b' − 'a' = 98 − 97 = 1 {ASCII value of a is 97 & of b is 98}

we can also use expressions like

(i) 'a' > 50 the result is True

(ii) 'a' > 'z' the result is True

(iii) 'a' > 'A' the result is True

String Manipulation

* Finding Length of a String

→ Length of a string is defined to be number of characters in it excluding null character.

→ C provides the built in function named strlen() to find length of string.

Syntax

```
int strlen(s);
```

where

int → Return type of function strlen
{ it returns the integer value }

strlen() → function name

s → string whose length is to be calculated

Working of Strlen

```
length=0;
for(i=0; str[i] != '\0'; i++)
{
    length++;
}
```

Program to find length of string

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
void main( )
{
    Char str[20];
    int i, length;
    clrscr( );
    printf ("Enter string");
    scanf ("%s", str);
    length= 0;
    for (i=0; str[i]!='\0'; i++)
      {
        length++;
      }
    printf("length of string is %d", length);
    length = strlen (str);
    printf ("Length of string using strlen() is %d", length);
    getch( );
}
```

* Copying One string to another string

→ Copying one string from string variable to another string variable is done by copying character by character from first variable to second variable.

→ This task cannot be performed by using assignment operator (=).

Syntax

```
strcpy (S2, S1);
```

Copies string contained in S1 to S2.

Note: S1 can be string constant or a string variable But S2 should be string variable & it should be large enough to collect the string in S1

Working of strcpy

```
for (i=0; str1[i]!='\0'; i++)
    str2[i] = str1[i];
    str2[i] = '\0';
```

Q. Program to copy one string to another string

```
# include<stdio.h>
# include <string.h>
# include <conio.h>
void main()
{
 Char str1[20]="abcd", str2[20], str3[20], str4[20];
 int i;
 clrscr();
 printf("String in str1 = %s \n", str1);
 for(i=0; str[i]!='\0'; i++)
  { str2[i] = str1[i];
  }
  str2[i]='\0';
 printf("String in str2 = %s \n", str2);

  strcpy(str3, str1);
  printf("String in str3 = %s \n", str3);
  strcpy(str4, "mno");
  printf("String in str4 = %s \n", str4);
   getch();
 }
```

* Comparing two strings

→ Two strings S1 & S2 cannot be compared using (==) operator, neither can be compared as greater or smaller using (<, >) operators.

→ Comparison can be done by comparing first pair of characters of S1 & S2, comparison of second pair of characters & so on

⇒ **Syntax**

```
int strcmp (S1, S2);
```

Note: The function return numeric difference between two strings. It returns positive value when S1 is alphabetically greater than S2,

It return negative value when S1 is alphabetically lower than S2,

It return 0 when S1 & S2 are equal.

Working of strcmp

```
i=0;
while ((str1[i]==str2[i])&&((str2[i]!='\0')&&
                                (str1[i]!='\0')))
  {  i++;
  }
diff = Str1[i] - Str2[i];
```

Program to compare two strings

```
# include < stdio.h >
# include < string.h>
# include < conio.h >
void main()
{
char str1[20] = {"abc"}, str2[20] = {"aac"},
                                    str3[20] = {"abc"};
int i, diff;
clrscr();
printf (" str1 = %s , str2 = % s \n", str1, str2);
i=0;
while((str1[i] == str2[i])&& (str1[i]!='\0') && (str2[i]!='\0'))
  { i++;
  }
diff= str1[i] - str2[i] ;
printf (" Difference between two string is %d \n", diff);
diff = strcmp (str1, str2) ;
printf (" Differenle between str1 & str2 is %d \n", diff);
diff = strcmp( str2, str1);
printf (" Difference between str2 & str1 is %d \n", diff);
diff = strcmp (str1, str3);
printf (" Difference between str1 & str3 is %d \n", diff);
getch();
}
```

*Concatenation of Two Strings

→ The process of appending one string to the end of another string is called concatenation.

→ If S_1 & S_2 are two strings and when S_2 is appended to the end of S_1, S_1 & S_2 are said to be Concatenatio

→ The string S_1 then contains its original string plus string contained in S_2

⇒ **Syntax :**

strcat (S_1, S_2)

It appends S_2 to S_1, S_2 can be string variable or string constant. But S_1 should string variable & large enough to collect both strings combined.

Working of strcat

```
for (i=0; str1[i]!='\0'; i++)
for (j=0; str2[j]!='\0'; j++)
str1[i+j] = str2[j];
str1[i+j] = '\0';
```

Q. Program to Concatenate two Strings

```
# include <stdio.h>
# include <string.h>
# include <conio.h>
void main()
  {
     Char str1[20]={"abc"}, str2[20]={"def"}
      Char  str3[20]={"ghi"};
      int i, j;
      clrscr();
      printf("str1=%s str2=%s \n", str1, str2);
    for (i=0; str1[i]!='\0'; i++)
      for (j=0; str2[j]!='\0'; j++)
        str1 [i+j] = str2[j];
         str1 [i+j] = '\0';
   printf("After Concatenation str1=%s\n", str1);
  strcat ( str1, str3) ;
   printf ("str1=%s\n", str1)
   strcat ( str1, "jkl");
    printf ("str1 = %s\n", str1);
        getch();
  }
```

Q. Program to find the position of one string in another string

```
# include < stdio.h>
# include < string.h>
# include < conio.h >
# include < process.h>
Void main()
{
  Char str1[20], str2[20];
   int len_str1, len_str2, i, j, flag;
  clrscr();
 printf ("Enter first string \n");
  scanf ("%s", str1);
 printf ("Enter second string\n");
  scanf ("%s", str2);
     len_str1 = strlen (str1);
     len_str2 = strlen (str2);
   if (len_str2 > len_str1)
    {
      printf ("%s can not be found in %s", str2,
                                          str1);
      exit (0);
    }
   for (i=0; i< (lenstr1- lenstr2+1); i++)
    { flag = 1;
```

```
for (j=0; j< len_str2; j++)
    if (str1[i+j] != str2[j])
    {
        flag=0;
        break;
    }

    if (flag)
        break;
    }
    if (flag)
        printf("%s is not found at position %d", str2,
                                                 str1, i);
    else
        printf("%s is not found in %s", str2, str1);
        getch();
    }
```

Structure:

→ In C, a structure is derived data type consisting of a collection of member elements & their datatype.

→ A variable of structure type is the name of a group of one or more members which may or may not be of the same data type.

Definition of Structure Templates

```
struct tag-name
{
    datatype member1;
    datatype member2;
    :
    datatype member n;
};
```

where

struct → is a keyword
tag-name → name of structure
datatype → simply a data type of elements
member → members of structure

Declaration of Structure Variable

```
Structure emp
  {
     int empno ;
     char name[20];
     float salary;
  };
```

now empno, name, salary are logically grouped.

→ To declare variable of structure emp

```
struct emp e;
```

e.empno	e.name	e.salary

→ All three data items share the common name e & are distinguishable by members empno, name & salary

→ The operator dot (·), called member operator, enables us to access each data item seprately

→ It is possible to declare variable of structure, while defining the structure.

Syntax

```
struct tag-name
  {datatype member1
     ⋮
  } Variable1, Variable2 ;
```

eg:

```
struct emp
    { int empno;
      char name[20];
      float salary;
    } e1, e2;
```

Taking values to variable of structure

*
```
e.empno = 121;
strcpy (e.name, "Raghav");
e.salary = 12000;
```

* using scanf

```
scanf ("%d %s %f", & e.empno, e.name, & e.salary)
```

* Initilization at the time of declaration

```
struct emp e = {121, "Anuj", 20000};
```

Operations on Structure

① Assigning value to individual structure variable's element

```
struct emp e;
e.empno = 10;
```

② Assigning one structure variable to another

```
struct emp e1 = {12, "Ram", 29000}, e2;
e2 = e1;
```

③ Retrieving the size of structure variable

```
struct emp e;
int s;
s = sizeof(e);
```

④ Retrieving the address of a structure

```
struct emp e;
&e
```

⑤ Checking whether two structure variables are equal

```
e1 == e2.
```

→ the expression returns 1 if both are equal

→ the expression returns 0 if both are unequal

Program to show concept of structure & some operation

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

struct emp
 {
   int empno;
   Char name[20];
   float salary;
 };

void main( )
  {
    struct emp e1, e2;
     int size;
     clrscr( );
     printf ("Enter empno, name & salary \n");
     scanf ("%d %s %f", & e1. empno, e1.name, & e1. salary);
     size = sizeof (e1);
   printf ("\n No of bytes required for e1 = %d \n \n", size);
      e2 = e1;
    printf ("After assigning e1 to e2 \n \n");
    printf ("e2. empno = %d \n", e2. empno);
    printf ("e2. name = %s \n", e2. name);
    printf ("e2. salary = %f \n \n", e2. salary);
     printf ("Address of e1 = %u \n", & e1);
   } getch( )
```

Output

Enter empno, name and salary
123 Nishu 34690

No. of bytes required for e1 = 26

After assigning e1 to e2

e2.empno = 123
e2.name = Nishu
e2.salary = 3469.00

Address of e1 = 65498

Array of Structure

An array of structure is declared that can hold the information of employees (empno, name & salary)

To store details of 10 employees we declare array of size 10

```
struct emp e[10];
```

here the array elements e[0], e[1], ---- e[9] are variables of struct emp type

Program to show array of structure

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

struct emp
    {
      int empno;
      char name [20];
      float salary;
    };
Void main()
    {
      struct emp e[10];
      int i, n;
      clrscr();
      printf ("Enter number of employees\n");
      scanf ("%d", &n);
      printf ("Enter employee details\n");
      for (i=0; i<n; i++)
      scanf ("%d %s %f", &e[i].empno, e[i].name,
                                  &e[i].salary);
      printf ("\n Employee's details\n");
      for (i=0; i<n; i++)
      printf ("\n %d %s %f", e[i].empno, e[i].name,
                                  &e[i].salary);
    } getch();
```

To search for an employee in list of employees

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

struct emp
{ int empno;
  Char name [20];
  float salary;
}
void main( )
{
  struct emp e[10];
  int i, n, flag;
  Char sname [20];
  clrscr();
  printf (" Enter No. of employee\n");
  scanf ("%d", &n);
  printf ("Enter employee's Details\n");
  for (i=0; i<n; i++)
  scanf ("%d %s %f", &e[i].empno, e[i].name,
                                  & e[i]. salary);
  printf (" Enter name of employee to be search\n");
      scanf ("%s", sname);
```

```
printf("%d employee's details \n", n);
    for (i=0; i<n; i++)
        printf("\n %d %s %f \n", e[i].empno, e[i].name,
                                e[i].salary);

printf("\n Name of employee to be searched \n");
 printf("\n %s \n", sname);

 flag = 0;
  for(i=0; i<n; i++)
    if (strcmp (e[i].name, sname) == 0)
      {  flag=1;
         break;
      }
    if (flag==1)
      printf ("found");
    else
      printf ("Not found");
      getch();
  }
```

Array within structure :

→ It is possible to use an array of integer as a member of structure.

Syntax

```
struct student
{
    int regno;
    char name[20];
    int marks[5];
}
```

Ex Program to create list of students details & display them.

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

struct student
{
    int reg_no;
    char name[20];
    int marks[5];
    int total;
    float percent;
};
```

```
void main()
{
    struct student s[10];
    int i, n, j;
    clrscr();
    printf("Enter no. of students \n");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
      printf("Enter reg.no, name of student %d \n", i+1);
      scanf("%d %s", & s[i].regno, s[i].name);
      printf("Enter marks of five subject of %s \n", s[i].name);
      for (i=0; i<n; i++)
      {for (j=0; i<5; j++)
        scanf("%d", & s[i].marks[j]);
      }
    for (i=0; i<n; i++)
    {
      s[i].total =0;
      for(j=0; j<5; j++)
      s[i].total += s[i].marks[j];
      s[i].percent = (float) s[i].total/5;
    }
```

```
printf("\n Reg-No  Name  Percentage\n");
    for (i=0; i<n ; i++)
      printf ("%d %s %f\n", s[i].reg-no, s[i].name,
                                    s[i].percent);
      getch();
    }
```

Structure within Structure

→ It is possible to use a structure as the element of another structure. This is a concept of nesting one structure within another structure.

eg.

```
Struct date
  {
   int day, month, year;
  };

Struct emp
  {
   int empno;
   char name[20];
   struct date doj;
   float salary;
  };
```

Program to show structure within structure

```
# include <stdio.h>
# include <conio.h>
struct date
  {
    int day, month, year;
  }

struct emp
  {
    int empno;
    Char name [20];
    struct date doj;
    float salary;
  }
,
void main( ).
  {
    struct emp e;
    clrscr();
    printf (" Enter empno, name, doj & salary \n");
    scanf ("%d %s %d %d %f",&e.empno, e.name, &e.doj.day, &e.doj.month,
                                  &e.doj.year, &e.salary);
    printf ("Empno= %d\n", e.empno);
    printf ("Name = %s\n", e.name);
    printf ("DOJ = %d %d %d \n", e.doj.day, e.doj.month, e.doj.year);
    printf ("Salary = %f \n", e.salary);
  }
```

Structure & Function:

1) Passing member of a structure Individually

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

struct student
    {
      int reg_no;
      char name [20];
      float percent;
    };
void display (int, char[], float);
void main( )
  {
     struct student s;
      clrscr().
      printf ("Enter Reg.No, Name & percentage\n");
      scanf ("%d %s %f", & s. reg-no, s.name, & s.percent);
     display (s. regno, s.name, s. percent);
      getch();
  }
void display (int reg.no, char name [], float percent)
{
printf ("%d %s %f", reg_no, name, percent);
}
```

2. Passing Entire structure at once

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

struct student
 { int reg_no;
   char name[20];
   float percent;
 };

void display (struct student);
void main()
 {
   struct student s;
   clrscr();
   printf (" Enter reg_No, Name & percentage \n");
   scanf ("%d %s %f", &s.reg_no, s.name, &s.percent);
   display (s);
   getch();
 }

void display (struct student s)
{ printf ("%d %s %f", s.reg_no, s.name, s.percent);
}
```

③ Passing array of structure to function

```
# include <stdio.h>
# include <conio.h>
struct emp
  { int empno;
    Char name[20];
    float salary;
  };
Void display (struct emp[], int);
 Void main()
    { struct emp e[10];
      int i,n;
      clrscr();
      printf("Enter No. of Employees");
       scanf("%d", &n);
     printf("Enter %d employees details\n", n);
       for(i=0; i<n; i++)
     scanf("%d%s%f", &e[i].empno, e[i].name, e[i].salary);
    printf("The list of employees\n");
       display(e,n);
    }getch();
 Void display (struct emp e[], int n)
   { int i;
     for(i=0; i<n; i++)
    printf("%d%s%f", e[i].empno, e[i].name, e[i].salary);
  }
```

Union :

⇒ The concept of union is derived from the concept of structure, The common thing shared by both structure and union is that both enable us to identify a group of data items which may be of different type of a common names.

⇒ The difference lies in their storage allocation scheme. In case of structure, the number of locations allocated would be equal to the number of members in structure.

In case of union, only one location which is large enough to collect largest data type member in the union gets allocated. This single location can accomodate values of different type at different types one at a time.

Syntax

```
union tag_name
{
    data type member1;
    data type member2;
    :
    :
    data type member n;
};
```

where:

Union is keyword

tagname is name of Union

datatype . type of data

member1, member2--- are members of Union

eg

```
Union temp
{
    int i;
    float f;
    char c;
};
Union temp t;
```

Q. To differentiate between structure & Union

```
# include <stdio.h>
# include <conio.h>
struct temp
{ int i;
  float f;
  char c;
};
union temp1
{ int i;
  float f;
  char c;
```

```
void main()
{ struct temp st;
    Union temp1 ut;
printf("size of structure is %d \n", sizeof(st));
printf("size of union temp = %d \n", sizeof(ut));
  st.i=10;
  st.f=3.45;
  st.c='s';
printf("st.i= %d \n", st.i);
printf("st.f= %f \n", st.f);
printf("st.c=%c \n", st.c);
  ut.i=10;
  ut.f=3.45;
  ut.c='s';
printf("ut.i= %d \n", ut.i);
printf("ut.f= %f \n", ut.f);
printf("ut.c=%c \n", ut.c);
}
```

Output

```
Size of structure is 7
size of union temp=4
st.i=10
st.f=3.450000
st.c=   s
ut.i= 14735
ut.f= 14.10001
ut.c=
```

Enumerated Data Type

Enumerated data type offers a way of inventing our own data type.

The invented data type enables us to assign symbolic names to integer constants thereby increasing the degree of readability & maintainability of program.

Syntax

```
enum tag_name
{
    enumerator-1;
    enumerator-2;
    enumerator-3;
        ⋮
    enumerator-n;
};
```

where

enum → a keyword

tag_name → user defined name

enumerator-1, enumerator-2 → symbolic name representing integer constant 0, 1 etc by default.

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

18CSS101J - Programming for Problem Solving Unit V

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

COURSE LEARNING RATIONALE (CLR)	*The purpose of learning this course is to:*
CLR -1:	Think and evolve a logically to construct an algorithm into a flowchart and a pseudocode that can be programmed
CLR -2:	Utilize the logical operators and expressions to solve problems in engineering and real-time
CLR -3:	Store and retrieve data in a single and multidimensional array
CLR -4:	Utilize custom designed functions that can be used to perform tasks and can be repeatedly used in any application
CLR -5:	Create storage constructs using structure and unions. Create and Utilize files to store and retrieve information
CLR -6:	Create a logical mindset to solve various engineering applications using programming constructs in C

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

COURSE LEARNING OUTCOMES (CLO)	*At the end of this course, learners will be able to:*
CLO - 1:	Identify methods to solve a problem through computer programming. List the basic data types and variables in C
CLO - 2:	Apply the logic operators and expressions. Use loop constructs and recursion. Use array to store and retrieve data
CLO - 3:	Analyze programs that need storage and form single and multi-dimensional arrays. Use preprocessor constructs in C
CLO - 4:	Create user defined functions for mathematical and other logical operations. Use pointer to address memory and data
CLO - 5:	Create structures and unions to represent data constructs. Use files to store and retrieve data

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

LEARNING RESOURCES	
S. No	**TEXT BOOKS**
1.	*Zed A Shaw, Learn C the HardWay: Practical Exercises on the Computational SubjectsYou Keep Avoiding (Like C),AddisonWesley, 2015*
2.	*W. Kernighan, Dennis M. Ritchie,The C Programming Language, 2nd ed. Prentice Hall, 1996*
3.	*Bharat Kinariwala,Tep Dobry, Programming in C, eBook*
4.	*http://www.c4learn.com/learn-c-programming-language/*

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

UNIT V

INTRODUCTION

Initializing Structure, Declaring Structure variable- Structure using typedef, Accessing members - Nested structure Accessing elements in a structure array - Array of structure Accessing elements in a structure array -Passing Array of Structure to function- Array of Pointers to structures- Bit Manipulation of structure and pointer to structure - Union Basic and declaration - Accessing Union Members Pointers to union

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY,
DELHI-NCR CAMPUS.

UNIT V

INTRODUCTION

Dynamic memory allocation, malloc, realloc , free - Allocating Dynamic Array- Multidimensional array using dynamic memory allocation- file: opening, defining, closing, File Modes, File Types- Writing contents into a file - Reading file contents - Appending an existing file- File permissions and rights- changing permissions and rights.

INTRODUCTION TO STRUCTURE

•Problem:

– How to group together a collection of data items of different types that are logically related to a particular entity??? (**Array**)

Solution: ***Structure***

STRUCTURE

- A Structure is a collection of variables of different data types under a singlename.
- The variables are called members of the structure.
- Thestructure is also called auser-defined data type.

Defining a Structure

•Syntax:

struct structure_name

{

data_type member_variable1; data_type member_variable2;

……; data_type member_variableN;

};

Once structure_name is declared asnew data type, then variables of that typecan be declared as:

struct structure_name structure_variable;

Note: The members of a structure do not occupy memory until they are associated with a structure_variable.

- Example

```
struct student
{
char name[20];  int roll_no;  float marks;  char
   gender;
long int phone_no;
};
struct student st;
```

- Multiple variables of *struct student* type can be declared as:

```
struct student st1, st2, st3;
```

Defining a structure…

- Each variable of structure has its own member copy of variables.
- The member variables are accessed using the dot (.) operator or memberoperator.
- For example: *st1.name* is member variable *name* of *st1* structure variable while *st3.gender* is member variable *gender* of *st3* structure variable.

Defining a structure…

```
struct student
{
char name[20];
int roll_no;  float
marks;  char gender;
long intphone_no;
}st1, st2, st3;
```

```
struct
{
char name[20]; int
roll_no; floatmarks;
char gender;
long intphone_no;
}st1, st2,st3;
```

Structure initialization

- Syntax:

struct structure_name structure_variable={value1, value2, …, valueN};

- Note: C does not allow the initialization of individual structure members within the structure definition template.

```
struct student
        {
    char name[20];  int roll_no;  float marks;  char gender;
    long int phone_no;
    };
    void main()
    {
    struct  student  st1={“ABC",  4, 79.5, 'M',  5010670};
    clrscr();
    printf("Name\t\t\tRoll No.\tMarks\t\tGender\tPhone No.");
    printf("\n.....\n");
    printf("\n %s\t\t %d\t\t %f\t%c\t %ld",        st1.name, st1.roll_no,
    st1.marks,
    st1.gender, st1.phone_no);
    getch();
    }
```

Initialization

Partial Initialization

- We can initialize the first few members and leave the remaining blank.
- However, the uninitialized members should be only at the end of the list.
- The uninitialized members are assigned default values as follows:
  - Zero for integer and floating point numbers.
  - '\0' for characters andstrings.

```
struct student
    {
  char name[20];  int roll;
  char remarks;
  float marks;
  };
  void main()
  {
  struct student s1={“name",
  4};
  clrscr();
  printf("Name=%s", s1.name);
  printf("\n Roll=%d", s1.roll);
  printf("\n Remarks=%c",
  s1.remarks);  printf("\n
  Marks=%f", s1.marks);
  getch();
  }
```

Accessing member of structure/ Processing a structure

- By using dot (.) operator or period operator or member operator.
- Syntax:

structure_variable.member

- Here, *structure_variable* refers to the name of a *struct* type variable and *member* refers to the name of a member within thestructure.

Question

•Create a structure named *student* that has *name*, *roll and mark* as members. Assume appropriate types and size of member. Write a program using structure to read and display the data entered by the user.

```
struct student
{
char name[20];
int roll;
float mark;
};

void main()
{
struct student s;
clrscr();
printf("Enter name:\t");  gets(s.name);
printf("\n Enter roll:\t");
scanf("%d", &s.roll);  printf("\n Enter
marks:\t");  scanf("%f", &s.mark);
printf("\n Name \t Roll \t Mark\n");
printf("\n.................................\n");
printf("\n%s\t%d\t%f", s.name, s.roll,
s.mark);
getch();
}
```

Copying and Comparing Structure Variables

- Two variables of the same structure type can be copied in the same way as ordinary variables.
- If *student1* and *student2* belong tothe same structure, then the following statements are valid:

student1=student2; student2=student1;

- However, the statements such as:

student1==student2 student1!=student2

are not permitted.

- If we need to compare the structure variables, we may do so by comparing members individually.

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

Here, structure has beendeclared global i.e. outside of main() function. Now, any function can access it and create a structure variable.

```
structstudent
                    {
char name[20];
int roll;
};
void main()
{
struct student
student1={“ABC", 4, };
struct student student2;
clrscr();  student2=student1;
printf("\nStudent2.name=%s",
student2.name);
printf("\nStudent2.roll=%d",
student2.roll);
if(strcmp(student1.name,student
2.na  me)==0 &&
(student1.roll==student2.roll))
{
printf("\n\n student1 and
student2  are same.");
}
getch();
}
```

How structure elements are stored?

contiguous

- The elements of a structure are always stored in memory locations.
- A structure variable reserves number of bytes equal to sum of bytes needed to each of its members.
- Computer stores structures using the concept of "word boundary". In a computer with two bytes word boundary, the structure variables are stored left aligned and consecutively one after the other (with at most one byte unoccupied in between them called slack byte).

How structure elements are stored?

- When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined.
- Due to this, even if the members of two variables are equal, their structures do not necessarily compare.

Array of structure

- Let us consider we have a structure as:

```
struct student
{
char name[20];
int roll;
char remarks;  float marks;
};
```

- If we want to keep record of 100 students, we have to make 100 structure variables like st1, st2, …,st100.
- In this situation we can use array of structure to store the records of 100 students which is easier and efficient to handle (because loops can be used).

Array of structure…

- Two ways to declare an array of structure:
- *struct student*
  - *{*
  - *char name[20];*
    int roll;
  - *char remarks;*
    float marks;
  - *}st[100];*

```
struct student
{
char name[20];
int roll;
char remarks;
float marks;
};
struct student
st[100];
```

- Write a program that takes roll_no, fname lname of 5 students and prints the same records in ascending order on the basis of roll_no

Reading values

```
for(i=0; i<5; i++)
{
printf("\n Enter roll number:");
 scanf("%d",  &s[i].roll_no);

printf("\n Enter first name:");
 scanf("%s",  &s[i].f_name);

printf("\n Enter Lastname:");
 scanf("%s",  &s[i].l_name);
}
```

Sorting values

```
for(i=0; i<5; i++)
{
for(j=i+1; j<5; j++)
{
if(s[i].roll_no<s[j].roll_no)
{
temp = s[i].roll_no;  s[i].roll_no=s[j].roll_no;
s[j].roll_no=temp;
}
}
}
```

Question

- Define a structure of employee having data members name, address, age and salary. Take the data for n employees in an array and find the average salary.
- Write a program to read the *name*, *address*, and *salary* of 5 employees using array of structure. Display information of each employee in alphabetical order of theirname.

Array within Structure

- We can use single or multi dimensional arrays of type *int* or *float*.

- Eg.

```
struct student
{
char name[20];
int roll;
float marks[6];
};
struct student s[100];
```

- Here, the member *marks* contains six elements, *marks[0]*, *marks[1]*, .., *marks[5]* indicating marks obtained in six different subjects.
- These elements can be accessed using appropriate subscripts.
- For example, *s[25].marks[3]* refers to the marks obtained in the fourth subject by the 26th student.

Array within structure…

Reading Values

```
        for(i=0;i<n;i++)
{
printf("\n Enter information about student%d",i+1); printf("\n
Name:\t");
scanf(" %s",  s[i].name); printf("\n
Class:\t");    scanf("%d", &s[i]._class);
printf("\n Section:");      scanf(" %c",  &s[i].section);
printf("\n Input marks of 6subjects:\t");     for(j=0;j<6;j++)
{
                        scanf("%f", &temp);
s[i].marks[j]=temp;

}        }
```

Structure within anotherStructure (Nested Structure)

- Let us consider a structure *personal_record* to store the information of a personas:

```
•struct personal_record
{
char name[20];int day_of_birth;
int month_of_birth; int year_of_birth;  float salary;
}person;
```

Structure within anotherStructure (Nested Structure)…

•In the structure above, we can group all the items related to birthday together and declare
them under a substructure as:

```
struct Date
{
int day_of_birth; int month_of_birth; int  year_of_birth;
};
                                    struct personal_record
{
char name[20];      struct Date birthday; float
salary;
```

Structure within another Structure (Nested Structure)…

- Here, the structure *personal_record* contains a member named *birthday* which itself is a structure with 3 members. This is called structure within structure.
- The members contained within the inner structure canbe accessed as:

person.birthday.day_of_birth person.birthday.month_of_birth
person.birthday. year_of_birth

- The other members within the structure are accessed as personal_record usual:

person.name *person.salary*

```
printf("Enter name:\t");
scanf("%s", person.name);
printf("\nEnter day of
birthday:\t");
scanf("%d",
&person.birthday.day_of_birth);
printf("\nEnter month of birthday:\t");
scanf("%d",
&person.birthday.month_of_birth);
printf("\nEnter year of birthday:\t");
scanf("%d",
&person.birthday.year_of_birth);
printf("\nEnter salary:\t");
```

Structure within anotherStructure (Nested Structure)…

- ***Note:- More than one type of structures can be nested…***

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

    struct name
    {
    char first_name[10];
    char middle_name[10];
    char last_name[10];
    };

    struct personal_record
    {
    float salary;
    struct date
    birthday,deathday;
    struct name full_name;
    };
```

Assignment

- Create a structure named ***date*** that has ***day***, ***month*** and ***year*** as its members. Include this structure as a member in another structure named ***employee*** which has ***name***, ***id*** and ***salary*** as other members. Use this structure to read and display employee's name, id, date of birthday andsalary.

Pointer to Structure

•Astructure type pointer variable can be declared as:

```
struct book
{
char name[20];
int pages; float price;
};
struct book *bptr;
```

•However, this declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer, so that to access structure's members through pointer ***bptr,*** we must allocate the memory using ***malloc()***function.

OR

•Now, individual structure members are accessed as:

bptr->name ***bptr->pages*** ***bptr->price***

(*bptr).name ***(*bptr).pages*** ***(*bptr).price***

•Here, **->** is called arrow operator and there must be a pointer to the structure on the left side of this operator.

```
struct book*bptr;

bptr=(struct book *)malloc(sizeof(struct
book));

printf("\n Enter name:\t");  scanf("%s",
bptr->name);  printf("\n Enter no. of
pages:\t");
scanf("%d", &bptr->pages);  printf("\n
Enter price:\t");  scanf("%f", & bptr-
>price=temp)
```

Pointer to Structure…

•Also, the address of a structure type variable can be stored in a structure type pointer variableasfollows:

```
struct book
{
char name[20]; int pages;  float price;
};
struct book b,*bptr;      bptr=&b;
```

•Here, the base address of *b* is assigned to*bptr* pointer.

Pointer to Structure…

- Now the members of the structure book can be accessed in 3 ways as:

b.name	*bptr->name*	*(*bptr).name*
b.pages	*bptr->pages*	*(*bptr).pages*
b.price	*bptr->price*	*(*bptr).price*

Pointer to array of structure

- Let we have a structure asfollows:

```
struct book
{
char name[20]; int pages;  float price;
};
struct book b[10], *bptr;
```

- Then the assignment statement *bptr=b;* assigns the address of the zeroth element of *b* to *bptr*.

Pointer to array of structure…

- The members of *b[0]* can be accessedas:

bptr->name bptr->pages bptr->price

- Similarly members of *b[1]* can be accessedas:

(bptr+1)->name (bptr+1)->pages (bptr+1)->price

- The following *for* statement can be used to print all the values of array of structure *b* as:

```
for(bptr=b;bptr<b+10;bptr++)
printf("%s %d %f", bptr->name, bptr->pages, bptr->price);
```

Problem

- Define a structure of employee having data members name, address, age and salary. Take data for n employee in an array **dynamically** and find the averagesalary.
- Define a structure of student having data members name, address, marks in C language, and marks in information system. Take data for n students in an array dynamically and find the total marksobtained.

Function and Structure

- We will consider four caseshere:
  - ***Passingthe individual members to functions***
  - ***Passingwhole structure to functions***
  - ***Passingstructure pointer to functions***
  - ***Passingarray of structure to functions***

Passing structure member to functions

- Structure members can be passed to functions as actual arguments in function call like ordinary variables.
- Problem: Huge number of structure members
- Example: Let us consider a structure *employee* having members *name*, *id* and *salary* and pass these members to afunction:

```
display(emp.name,emp.id,emp.salary);

Void display(char e[],int id ,float sal)
{
printf("\nName\t\tID\t\tSalary\n);
printf("%s\t%d\t%.2f",e,id,sal);
}
```

Passing whole structure to functions

- Whole structure can be passed to a function by the syntax:
- ***function_name(structure_variable_name);***
- The called function has the form:
- ***return_type function_name(struct tag_name structure_variable_name)***
  - {
  - ……;
  - }

display(emp);

```
void display(struct employee e)
{
printf("\nName\tID\tSalary\n");
printf("%s\t%d\t%.2f",e.name,e.id,e.salar);
}
```

Passing structure pointer tofunctions

- In this case, address of structure variable is passed as an actual argument to a function.
- The corresponding formal argument must be a structure type pointer variable.
- Note: Any changes made to the members in the called function are directly reflected in the calling function.

```
display(&emp);

void display(struct employee*e)
{
printf("\nName\tID\tSalary\n");
printf("%s\t%d\t%.2f",e->name,e->id,e->salary);
}
```

Passing array of structures to function

- Passing an array of structure type to a function is similar to passing an array of any type to afunction.
- That is, the name of the array of structure is passed by the calling function which is the base address of the array of structure.
- Note: The function prototype comes after the structure definition.

display(emp); //emp is array name of size2

```
void display(struct employee ee[])
{
int i;
printf("\n Name\t\t ID\t\t Salary\n");  for(i=0;i<2;i++)
{
printf("%s\t\t%d\t\t%.2f\n",ee[i].name,ee[i].id,ee[i].sala
ry);
}
}
```

Structure using typedef

- It allows us to introduce synonyms for data types which could have been declared some other way.
- It is used to give New name to the Structure.
- New name is used for Creating instances, Passing values to function,declaration etc…

Example:

```
#include<stdio.h>
int main()
{
typedef int Number;
Number num1 = 40,num2 =
20;
Number answer;  answer =
num1 + num2;
printf("Answer : %d",answer);
return(0);
}
```

Output :

Answer : 60

- In the above program we have used typedef to create alias name to data type. We have created alias name to ‘int’ data type. We have given new name to integer data type i.e ‘Number’.

- In the second example, **Record** is **tag-name**. **’employee’** is nothing but **New Data Type**. We can now create the variables of type **’employee’** Tag name is optional.

Different Ways of Declaring Structure using Typedef :

```
typedef struct
{
char ename[30];
int ssn;
int deptno;
 }
employee;
```

```
typedef struct Record
 {
char ename[30];
 int ssn;
 int deptno;
}
employee;
```

Live Example : Using Typedef For Declaring Structure

```
#include<stdio.h
>
typedef struct
b1
{
char bname[30];
int ssn;
int pages;
}
book;
book b1 = {"Let
Us  C",1000,90};
int main()
{
printf("\nName of Book
:s",b1.bname);
printf("\nSSN of Book : %d",b1.ssn);
printf("\nPages in Book :
%d",b1.pages);  return(0);
}
```

OUTPUT:

```
Name of Book : Let Us C  SSN of
Book : 1000
Pages in Book : 90
```

Pointer to Structure Array

- Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently,
- we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

Accessing Structure Members with Pointer

```
#include <stdio.h>
struct Book
{ char name[10];  int price; }
int main()
{
struct Book a; //Single structure variable struct Book* ptr; //Pointer of
Structure type ptr = &a;  struct Book b[10]; //Array of structure
variables struct Book* p; //Pointer of Structure type
p = &b;
return 0;
}
```

Accessing Structure Members with Pointer

- To access members of structure using the structure variable, we used the dot . operator.

- But when we have a pointer of structure type, we use arrow -> to access structure members.

Accessing Structure Members with Pointer

```
#include <stdio.h>  struct my_structure {  char name[20];
int number;  int rank;
};
int main()
{
struct my_structure variable = {"StudyTonight", 35, 1}; struct my_structure *ptr; ptr = &variable;
printf("NAME: %s\n", ptr->name);
printf("NUMBER: %d\n", ptr->number);
printf("RANK: %d", ptr->rank);
return 0;
}
```

Bit Manipulation

•Suppose we want to store the gender of the person , then instead of wasting the complete byte we can manipulate single bit. We can consider Single bit as a flag. Gender of Person can be stored as **[M/F]** . By Setting the **flag bit**

•**1** we can set Gender as **"Male"** and "**Female**" by setting bit as 0

•1. To pack **Several data objects** in the single memory word , Bits are used.

•2. Flags can be used in order to store the Boolean values **(T / F)**.

•3. A method to define a structure of packed information is known as **bit fields**

Syntax : Bit Manipulation

```
struct databits
{
int b1 : 1;  int b2 : 1;  int b3 : 1;  int
b4 : 4;  int b5 : 9;
}data1;
```

Explanation :

- In the above example, we can say that we have **allocated specific number of bits to each structure member**.

- Integer can store 2 bytes (*) , so 2 bytes are **manipulated in bits** and 1 bit is reserved for b1. Similarly b2,b3 will get single bit.

- Similarly we can structure efficiently to **store boolean values or smaller values** that requires little membory.

How to access the Individual Bits ?

- data1.b1
- data1.b2
- data1.b3
- data1.b4
- data1.b5

How to initialize Structure ?

```
struct databits
{
- -
- - -
}data1 = { 1,1,0,10,234 };
```

Initialized Result -

```
data1.b1 = 1
data1.b2 = 1
data1.b3 = 0
data1.b4 = 10
 data1.b5 = 234
```

Unions

- Unions are quite similar to the structures in C.
- Union is also a derived type as structure.
- Union can be defined in same manner as structures just the keyword used in defining union in union where keyword used in defining structure was struct.

syntax

```
union car
{
char name[50];
int price;
};
```

Union variables can be created in similar manner as structure variable.

```
union car
{
char name[50];
int price;
}c1, c2, *c3;
```

OR

```
union car
{
char name[50];
```

Accessing members of an union

•The member of unions can be accessed in similar manner as that structure. Suppose, we you want to access price for union variable c1 in above example, it can be accessed as c1.price. If you want to access price for union pointer variable c3, it can be accessed as (*c3).price or as c3->price.

Example


```
#include
<stdio.h>
union job
{
char name[32];
float salary;
int worker_no;
}u;
struct job1
{
char name[32];
float salary;
int worker_no;
}s;
int main()
{
```

output

size of union = 32 size of structure = 40

Memory allocation

There is difference in memory allocation between union and structure as suggested in below example. The amount of memory required to store a structure variables is the sum of memory size of all members.

Fig: Memory allocation in case of structure

But, the memory required to store a union variable is the memory required for largest element of an union.

Fig: Memory allocation in case of union

Diff between structure and union

•All members of structure can be accessed at any time. But, only one member of union can be accessed at a time in case of union and other members will contain garbage value.

Example

```
#include <stdio.h>
union job
{
char name[32];  float salary;
int worker_no;
}u;
int main()
{
printf("Enter name:\n");
scanf("%s",u.name);
printf("Enter salary: \n");
scanf("%f",&u.salary);
printf("Displaying\nName
:%s\n",u.name);  printf("Salary:
%.1f",u.salary);
```

output

Enter name Hillary Enter salary 1234.23

Displaying Name: f%Bary Salary: 1234.2

Initially, Hillary will be stored in u.name and other members of union will contain garbage value. But when user enters value of salary, 1234.23 will be stored in u.salary and other members will contain garbage value. Thus in output, salary is printed accurately but, name displays some random string.

Accessing Union Members

•To access any member of a union, we use the **member access operator (.)**.

```
#include <stdio.h>
#include <string.h>  union Data
{
int i;
float f;
char str[20];
};
int main( )
{
union Data data; data.i = 10;
```

output

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Example

•Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions

–

```
#include <stdio.h>
#include <string.h>  union Data
{ int i;
float f;
char str[20];
};
int main( )
{
union Data data;  data.i = 10;
printf( "data.i : %d\n", data.i);
data.f = 220.5;
```

output

data.i : 10

data.f : 220.500000

data.str : CProgramming

Dynamic Arrays

- Dynamic arrays are created using pointer variables and memory management functions malloc,calloc and realloc.
- The concept of dynamic arrays is used in creating and manipulating data structures such as linked list, stacks and queues.

File Handling in C

What is a File?

- A *file* is a collection of related data that a computers treats as a single unit.
- Computers store files to secondary storage so that the contents of files remain intact when a computer shuts down.
- When a computer reads a file, it copies the file from the storage device to memory; when it writes to a file, it transfers data from memory to the storage device.
- C uses a structure called **`FILE`** (defined in **`stdio.h`**) to store the attributes of a file.

Steps in Processing a File

1. Create the stream via a pointer variable using the **FILE** structure:

```
FILE *p;
```

2. Open the file, associating the stream name with the file name.
3. Read or write the data.
4. Close the file.

The basic file operations are

- fopen - open a file- specify how its opened (read/write) and type (binary/text)
- fclose - close an opened file
- fread - read from a file
- fwrite - write to a file
- fseek/fsetpos - move a file pointer to somewhere in a file.
- ftell/fgetpos - tell you where the file pointer is located.

File Open Modes

Mode	Meaning
r	Open text file in read mode • If file exists, the marker is positioned at beginning. • If file doesn't exist, error returned.
w	Open text file in write mode • If file exists, it is erased. • If file doesn't exist, it is created.
a	Open text file in append mode • If file exists, the marker is positioned at end. • If file doesn't exist, it is created.

More on File Open Modes

(a) Read Mode (b) Write Mode (c) Append Mode

Additionally,

- r+ - open for reading and writing, start at beginning
- w+ - open for reading and writing (overwrite file)
- a+ - open for reading and writing (append if file exists)

File Open

- The file open function (`fopen`) serves two purposes:
  - It makes the connection between the physical file and the stream.
  - It creates “a program file structure to store the information” Cneeds to process the file.
- Syntax:

```
filepointer=fopen("filename", "mode");
```

More On `fopen`

- The file mode tells C how the program will use the file.
- The filename indicates the system name and location for the file.

fopen

spData = fopen("MYFILE.TXT", "w");
spData = fopen("A:\\MYFILE.TXT", "w");

- We assign the return value of **fopen** to our pointer variable:

```
spData = fopen("MYFILE.TXT", "w");
spData = fopen("A:\\MYFILE.TXT", "w");
```

More On fopen

```
#include <stdio.h>
…
{
int main (void)
  FILE* spData;
  …
  spData = fopen("MYDATA.DAT", "w");
  …
} // main
```

Internal File Variable

External File Name

Closing a File

- When we finish with a mode, we need to close the file before ending the program or beginning another mode with that same file.
- To close a file, we use `fclose` and the pointer variable:

```
fclose(spData);
```

fprintf()

Syntax:

Syntax:

fprintf (fp,"string",variables);

Example:

Example:

```
int i = 12;
float x = 2.356;  char ch = 's';  FILE *fp;
fp=fopen("out.txt","w");
fprintf (fp, "%d %f %c", i, x, ch);
```

fscanf()

Syntax:

Syntax:

fscanf (fp,"string",identifiers);

Example:

Example:

```
FILE *fp;
Fp=fopen(“input.txt”,”r”);  int i;
fscanf (fp,“%d",i);
```

getc()

Syntax:

Example:

Syntax:

identifier = getc (file pointer);

Example:

```
FILE *fp;
fp=fopen("input.txt","r");  char
ch;
ch = getc (fp);
```

putc()

write a single character to the output file, pointed to by fp.

Example:

Example: FILE *fp; char ch;

putc (ch,fp);

End of File

fscanf

•There are a number of ways to test for the end-of-file condition. Another way is to use the value returned by the *fscanf* function:

```
FILE *fptr1;  int istatus ;
istatus = fscanf (fptr1, "%d", &var) ;
if ( istatus == feof(fptr1) )
{
printf ("End-of-file encountered.\n”) ;
}
```

Reading and Writing Files

```
#include <stdio.h>  int main ( )
{
FILE *outfile, *infile ;  int b = 5, f ;
float a = 13.72, c = 6.68, e, g ;
outfile = fopen ("testdata", "w") ;
fprintf (outfile, “ %f %d %f ", a, b, c) ;
fclose (outfile) ;
infile = fopen ("testdata", "r") ;
fscanf (infile,"%f %d %f", &e, &f, &g) ;
printf (“ %f %d %f \n ", a, b, c) ;
printf (“ %f %d %f \n ", e, f, g) ;
```

Example

```
#include <stdio.h>
#include<conio.h> void
main()
{
char ch;  FILE *fp;
fp=fopen("out.txt","r");
while(!feof(fp))
{
ch=getc(fp);
printf("\n%c",ch);
}
getch();
}
```

fread ()

size_t fread(void *ptr, size_t size, size_t n, FILE *stream);

Declaration:

size_t fread(void *ptr, size_t size, size_t n, FILE *stream);

Remarks:

fread reads a specified number of equal-sized
data items from an input stream into a block.

ptr = Points to a block into which data is read, size = Length of each item

Example

Example:

```
#include <stdio.h>  int main()
{
FILE *f;
char buffer[11];
if (f = fopen("fred.txt", “r”))
{
fread(buffer, 1, 10, f);
buffer[10] = 0;  fclose(f);
printf("first 10 characters of the file:\n%s\n",
buffer);
}
return 0;
}
```

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY,
DELHI-NCR CAMPUS.

fwrite()

Declaration:

size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);

Remarks:

fwrite appends a specified number of equal-sized data items to an output file.

ptr = Pointer to any object; the data written begins at ptr

size = Length of each item of data

n =Number of data items to be appended

stream = file pointer

size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);

ptr = Pointer to any object; the data written begins at ptr

size = Length of each item of data

n =Number of data items to be appended

stream = file pointer

Example:

Example

Example:

```
#include <stdio.h>  int main()
{
char a[10]={'1','2','3','4','5','6','7','8','9','a'};
FILE *fs;
fs=fopen("Project.txt","w");  fwrite(a,1,10,fs);
fclose(fs);  return 0;
}
```

fseek()

This function sets the file position indicator for the stream pointed to by stream or you can say it seeks a specified place within a file and modify it.

SEEK_SET	Seeks from beginning of file
SEEK_CUR	Seeks from current position
SEEK_END	Seeks from end of file

Example:

```
#include
<stdio.h>

int main()
{FILE * f;
 f = fopen("myfile.txt",
 "w");  fputs("Hello
 World", f);
 fseek(f, 0,
 SEEK_SET);
 fputs(" India", f);
 fclose(f);
 return 0;
}
```

SEEK_CUR, SEEK_END

offset = ftell(file pointer);

ftell()

offset = ftell(file pointer);

"ftell" returns the current position for input or output on the file #include <stdio.h>

```
int main(void)
{
FILE *stream;
stream = fopen("MYFILE.TXT", "w");  fprintf(stream, "This is a test");
printf("The file pointer is at byte %ld\n", ftell(stream));
fclose(stream);
return 0;
}
```

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, DELHI-NCR CAMPUS.

THANK YOU……

Pointers:

→ A pointer is a variable that can hold the address of other variable, structure & function that are used in program.

→ It contains only the memory location of the variable rather than its content.

→ Pointers can be used to point to the variable such as variable of any basic data type, an array, a function, a structure and union.

Advantages:

* To point to different data structure
* manipulation of data at different memory location is easier
* To achieve ~~clarat~~ clarity & simplicity.
* More compact & efficient coding.
* To return multiple values via functions
* Dynamic memory allocation.

Operators Used with Pointers

There are two basic operators used with pointers

1) The address operator : & (ampersand)
2) The indirection operator : * (asterisk)
   (Value)

Pointer Declaration

→ When a pointer to any variable (of any type) is declared the compiler will allocate memory only for pointer, but not for type to which it points.

→ Both the pointer variable & the variable to which pointer points to, must be declared.

→ A pointer variable must be assigned the address of the declared variable before it is used in program.

Syntax

```
ptr = & ivar
```

where

ptr = a pointer variable
ivar = a variable

here ptr holds the address of variable ivar not value of ivar.

example

```
int x, y, *px, *py;
px = &x;
py = &y;
```

Program To print address & value of any variable

```
# include <stdio.h>
void main()
{
    int ivar=486;
    int * iptr;
    iptr= & ivar;
    printf ("Address of ivar = %d \n", iptr);
    printf ("The content of ivar = %d \n", * iptr);
}
```

Output

```
Address of ivar = 4056
The content of ivar = 486
```

Program to accept integer & floating value & print their value using pointer

```
# include <stdio.h>
void main()
{
    int x, *iptr;
    float y, *fptr;
    printf("Enter the value of x & y\n");
    scanf("%d %f", &x, &y);
    iptr = &x;
    fptr = &y;
    printf("The address of x= %d & its value is %d\n", iptr,*iptr);
    printf("The address of y=%d & its value is %f\n", fptr, *fptr);
}
```

<u>output</u>

```
Enter the value of x & y
25
66.5
The address of x=4056 & its value is 25
The address of y=4058 & its value is 66.500000
```

Pointer Dereferencing

→ The process of referencing the content of variable by pointers is called pointer dereferencing.

→ Steps involved in dereferencing

(a) Address of variable whose value is to be referenced is assigned to a pointer

(b) Then, this pointer points to the value.

Program to accept character & print is along with its address & value pointed by pointer

```
# include <stdio.h>
void main()
{
    Char alp, *cptr;
    printf ("Enter a character\n");
    alp = getchar();
    cptr = &alp;
printf ("The character is %c\n", alp);
printf (" Address of alphabet is %d\n", cptr);
    printf (" cptr points to %c\n", *cptr);
}
```

Operations on Pointers

1) Assignment

(a) A pointer variable can be assigned address of other variable

eg
```
int *ptr, var;
ptr = &var;
```

(b) If two pointer variables are pointing to the object of same data type they can be assigned

eg
```
int *ptr1, *ptr2;
ptr1 = ptr2;
```

(c) A pointer variable can be assigned a NULL

eg
```
int *ptr;
ptr = Null;
```

2) Addition & Substraction

(a) An integer value can be added to & substracted from a pointer variable

```
int *ptrx;
ptrx = (ptrx+2);
ptrx = (ptrx-1);
```

```
int *ptr, *ptr2, *ptr3;
ptr3 = ptr2 - ptr1;
```

Comparison

```
int *ptr1, *ptr2;
```

then, (ptr1 == ptr2)
(ptr1 != ptr2)
(ptr1 < ptr2)
(ptr1 > ptr2)

Pointers & Function

→ Like other variables pointers can also be passed to the function.

Call by Reference

→ In this mechanism, by which pointers can be passed as arguments to the function

When the pointers are passed as argument, then following two points must be considered

1) In the calling program, the function is invoked with the function name & address of actual parameter enclosed within paranthesis

ie function_name (&var1, &var2 - - - &varn)

2) In the called program, parameter list, each and every formal parameter (pointer) must be preceded by value operator (*)

ie datatype function_name (*var1, *var2 - - *varn)

<u>Program</u> to show call by reference

```
# include <stdio.h>
Void main()
{ int num1, num2;
  int inter_change (int *n1, int*2);
  printf ("Enter two numbers\n");
  scanf ("%d %d", &num1, &num2);
  printf ("num1 = %d and num2 = %d (before)\n", num1, num2);
  inter_change (&num1, &num2);   // function call
  printf ("num1 = %d & num2 = %d (after)\n", num1, num2);
}
int inter_change (int *n1, int *n2)
{ int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Pointers & Array

The name of an array itself designate some memory location & this location in the memory is the address of the very first element of the array. The address of first element of array can be written as:

& array[0] or simply array

address of second element

& array[1] & so on

Program to find sum of statically declared 5 elements using pointers & function

```
# include <stdio.h>
void main()
{
   int array[5] = {200, 400, 600, 800, 1000};
   int addnum (int *ptr);
   int sum;
   sum = addnum (array);
   printf ("Sum of all array elements = %d\n", sum);
}
int addnum (int *ptr)
{
   int total=0, i;
   for (i=0; i<5; i++)
      total += *(ptr + i);
   return (total);
}
```

Introduction to Dynamic Memory Allocation

→ In static memory allocation, size of memory may be required for the calculation that must be defined before loading & executing the program.

→ In case of dynamic memory allocation, the required memory size is allocated while program is executing

Memory Management Functions

The alloc family of functions are declared in the following header files

1) stdlib.h
2) alloc.h

a) Alloc ()

→ The alloc () function is used to allocate a region or block of size bytes in length of the heap.

→ The general declaration of alloc () is

```
char * alloc (size);
unsigned int size;
```

→ Zeros are initially allocate to every memory location allocated

Program code to illustrate usage of alloc()

```
struct list {
    int value;
    struct list * value;
};
struct list * ptr;
ptr = (struct list) * alloc(sizeof(struct list));
```

Malloc()

The malloc() function is used to allocate heap storage, Its name stands for memory allocation.
The allocated region is not filled with zeros.
The starting address is returned if function successful.
A zero is returned if function fails to get block of memory.

General declaration

```
Char *malloc (size)
unsigned int size;
```

eg:

```
struct list {
    int value;
    struct list * value;
};
struct list * ptr;
ptr = (struct list *) malloc(sizeof(Struct list));
```

Calloc( )

The calloc function is used to allocate continuous memory or element by element basis.

The name stands for calculated allocation.

It is useful for array & array of structure where continuity of memory is required

Starting address of area is returned if function is successful, A zero is returned if function fails to allocate block

General declaration

```
char * calloc (element, elementsize)
    int element;
    int element_size;
```

eg:

```
struct list {
        int value;
        struct list * value;
    };

struct list * ptr;
ptr = (struct list *) calloc (1, sizeof (struct list));
```

Realloc ( )

The realloc() is used to increase or decrease the size of block to the size specified by size while preserving the address & contents of beginning of the block.

The function realloc() can diminish size of allocated area

General declaration

```
char * realloc (old block, size)
char * old block ;
int size ;
```

Free ( )

The free() function is used to free a portion of storage within block previously allocated by alloc(), malloc(), calloc().

The storage is freed, making it available for further activities. The pointer that is the arguments to free() is the starting address of region allocated.

General declaration

```
int free (pointer)
char * pointer ;
```

File Handling

A file is defined to be a collection of related data stored on secondary storage device.

The concept of file enables us to store large amount of data permanently on the secondary storage device.

File handling means dealing with these files with the help of C language & perform operations on these files.

Opening & closing of Files

(i) fopen()

The fopen() is to open a file, basically establishing a link between a program & the file being opened.

Syntax

```
fp = fopen("filename", "mode of opening");
```

fp is filepointer which represents file
filename is name of file being opened
mode can be either w, r, a

Mode of opening	Purpose
'w'	To create a text file. If the file ~~exite~~ already exists, its contents are destroyed, otherwise it is created, if possible
'r'	To open a text file for reading (The file must exists)
'a'	To open a text file for appending (writing at end), if the file doesn't exists it is created.

(ii) fclose()

fClose is the counterpart of fopen(). This is to close a file. Closing a file means de-linking the file from the program & saving the content of the files

Syntax

```
fclose(fp);
```

fputc() & ~~fput~~ fgetc()

* fputc() is to write a character onto a file.

Syntax

```
fputc(c,fp);
```

where

c represents a character

fp represents filepointer to represent file

* fgetc() is to read a character from a file.

Syntax

```
c= getc(fp);
```

where

c is a variable of char type

fp is file pointer to represent a file

→ The function reads a character from the file denotes by fp & return character value which is collected by variable c.

Program to create file consisting of characters

```
# include < stdio. h>
# include< conio.h>
Void main()
  {
      FILE *fp;
      Char c;
      clrscr();
      fp= fopen("text","w");
      printf (" Keep typing. Type q to quit \n");
        c= getchar();
        while ( c!='q')
          {
            fputc ( c, fp);
              c= getchar();
            }
            fclose (fp);
      }
```

Program to Read a file consisting characters

```
# include <stdio.h>
# include <conio.h>
Void main()
{
    FILE *fp;
    Char c;
    clrscr();
    printf ("The content of file are\n");
    fp= fopen("text","r");
    while (!feof (fp))
    {
        c= fgetc (fp);
        putchar (c);
    }
    fclose (fp);
}
```

String oriented function

* fputs is used to write a string onto a file.

Syntax

fputs (buffer, fp);

where buffer is the name of character array & fp is pointer to FILE type.

→ The function writes the string represented by buffer to file poined by fp.

* fgets (buffer, size, fp);

where buffer is name of character array, size is an integer value, fp is a pointer to FILE

→ The function reads a string of size-1 character from file pointed by fp.

Program to create file consists of String

```
# include < stdio.h >
# include < conio.h >
void main()
{
    Char name [20];
    FILE *fp;
    int i, n;
    Clrscr();
    fp= fopen ("names.dat", "w");
    printf ("Enter the no. of names\n");
    scanf ("%d", &n);
    printf ("Enter %d names\n", n);
    for (i=1; i<=n; i++)
    {
        gets (name);
        fputs (name, fp);
    }
    fclose (fp);
}
```

Program to read a file consisting of string

```
# include <stdio.h>
# include <conio.h>
void main()
{
  FILE *fp;
  char name[20];
  clrscr();
  printf("strings are \n");
  fp = fopen ("names.dat", "r");
  while (!feof (fp))
  {
  fgets (name, 80, fp);
     puts (name);
       printf ("\n");
   }
     close (fp);
  }
```

Mixed Data Oriented Functions -

* fprint()

fprint is used to write multiple data items which may (or not) be of different types to a file.

Syntax

fprint (fp, " Control string", arguments-list);

* fscanf()

fscanf is to read multiple data items which may be of different type from a file.

Syntax

fscanf (fp, " control string", argument-list);

Program to create a file consisting of employee's detail

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

struct emp
{
  int empno;
  char name [20];
  float salary;
};

void main()
{
  FILE *fp;
  struct emp e;
  int i, n;
  fp = fopen ("emp.dat", "w");
  clrscr();
  printf ("enter no. of employees \n");
  scanf ("%d", &n);
  printf ("\n Enter %d employee's details \n", n);
  for (i=0; i<n; i++)
  {
    scanf ("%d %s %f", &e.empno, e.name,
                                  &e.salary);
  fprintf (fp, "%d %s %f", e.empno, e.name, e.salary);
  }
  fclose (fp);
}
```

Program to implement use of fwrite()

```
# include <stdio.h>
# include <conio.h>
struct emp
{
    int empno;
    char name[20];
    float salary;
};
void main()
{
    struct emp e;
    FILE *fp
    int i, n;
    clrscr();
    fp= fopen("emp.dat", "wb");
    printf ("Enter no. of employees\n");
    scanf ("%d", &n);
    printf ("\n Enter empno, name & salary of
                    %d employees\n", n);

    for (i=1; i<n; i++)
    {
        scanf ("%d %s %f", &e.empno, e.name,
                                &e.salary);
        fwrite (&e, sizeof(e), 1, fp);
    }
    fclose (fp);
    getch();
}
```

Program to implement use of fread()

```
# include <stdio.h>
# include < conio.h>
struct emp
  {
    int empno;
    char name [20];
    float salary;
  };
void main()
  {
    struct emp e;
    FILE *fp;
    int i,n;
    clrscr()
    printf ("content of file emp.dat \n");
    fp = fopen ("emp.dat", "rb");
    while (fread (&e, sizeof (e), 1, fp))
    {
    fprintf (fp, "%d %s %f \n", e.empno, e.name
                                ,e.salary);
    }
    fclose (fp);
    getch();
  }
```

Unformatted Record I/O Functions - fwrite(), fread()

* fwrite():

This function is used to write block of data (records) to a file.

Syntax

fwrite (buffer_address, size, count, file_pointer);

buffer_address → address of memory area, the contents of which are to be written to file

size → the number of bytes of a block (record)

Count → number of blocks of data to be written to the file

eg:

```
struct emp
  {
    int empno;
    char name[20];
    float salary;
  };
struct emp e;
FILE *fp;
fwrite (&e, sizeof(e), 1, fp);
```

* fread()

This function is counterpart of fwrite() & it is used to read block of data (record) from files.

syntax

```
fread (buffer_address, size, count, file pointer);
```

eg

```
struct emp
{
    int empno;
    char name [20]
    float salary;
}

struct emp e;
FILE *fp;
fread (&e, sizeof(e), 1, fp);
```

enum tag-name Variable; (General Syntax)

Eg :-

```
enum Boolean
{ false;
  true;
};

enum boolean f;
f = false;
```

Typedef =

→ Typedef provides a facility to rename existing built in data type & user defined data type & thereby increasing the degree of readability of source code.

Syntax

typedef oldname newname.

→ oldname is the name of existing datatype

→ new name is the new name given to datatype

eg:

```
typedef unsigned int CSE
```

now

```
CSE i;
```

⇒

```
struct emp
{ int empno;
  char name[20];
  float salary;
};

typedef struct emp EMP;
```

now

```
EMP e;
```