

---

# Programming for Problem Solving

1st Year 1st Semester 2022-23

Credits: 3

## Topics

### UNIT I: Introduction to Programming

- **Compilers, compiling and executing a program**
- **Representation of Algorithm**
  - Algorithms for finding roots of a quadratic equations
  - Finding minimum and maximum numbers of a given set
  - Finding if a number is prime number
  - Flowchart/Pseudocode with examples
  - Program design and structured programming
- **Introduction to C Programming Language**
  - Variables (with data types and space requirements)
  - Syntax and Logical Errors in compilation
  - Object and executable code
  - Operators, expressions and precedence
  - Expression evaluation
  - Storage classes (auto, extern, static and register)
  - Type conversion
  - The main method and command line arguments
  - Bitwise operations
    - Bitwise AND, OR, XOR and NOT operators
- **Conditional Branching and Loops**
  - Writing and evaluation of conditionals and consequent branching with if, if-else, switch-case, ternary operator, goto
  - Iteration with for, while, do- while loops
- **I/O**
  - Simple input and output with scanf and printf
  - Formatted I/O
  - Introduction to stdin, stdout and stderr
  - Command line arguments

### UNIT II: Arrays, Strings, Structures and Pointers

- **Arrays**
  - One and two dimensional arrays
  - Creating, accessing and manipulating elements of arrays

- 
- **Strings**
    - Introduction to strings
    - Handling strings as array of characters
    - Basic string functions available in C (strlen, strcat, strcpy, strstr etc.)
    - Arrays of strings
  - **Structures**
    - Defining structures
    - Initializing structures
    - Unions
    - Array of structures
  - **Pointers**
    - Idea of pointers
    - Defining pointers
    - Pointers to Arrays and Structures
    - Use of Pointers in self-referential structures
    - Usage of self-referential structures in linked list (no implementation)
  - Enumeration data type

## **UNIT III: Preprocessor and File handling in C**

- **Preprocessor**
  - Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef
- **Files**
  - Text and Binary files
  - Creating, Reading and Writing text and binary files
  - Appending data to existing files
  - Writing and reading structures using binary files
  - Random access using fseek, ftell and rewind functions

## **UNIT IV: Function and Dynamic Memory Allocation**

- **Functions**
  - Designing structured programs
  - Declaring a function
  - Signature of a function
  - Parameters and return type of a function
  - Passing parameters to functions
  - Call by value
  - Passing arrays to functions
  - Passing pointers to functions
  - Idea of call by reference
  - Some C standard functions and libraries
- **Recursion**
  - Simple programs, such as Finding Factorial, Fibonacci series etc.

- 
- Limitations of Recursive functions
  - **Dynamic memory allocation**
    - Allocating and freeing memory
    - Allocating memory for arrays of different data types

## UNIT V: Searching and Sorting

- **Basic searching in an array of elements**
  - Linear search technique
  - Binary search technique
- **Basic algorithms to sort array of elements**
  - Bubble sort algorithm
  - Insertion sort algorithm
  - Selection sort algorithms
- **Basic concept of order of complexity through the example programs**

# Question and Answers

1. Creating, compiling and executing a program.
2. What is algorithm, characteristics of algorithm and example.
3. Algorithm for finding roots of quadratic equation.
4. Algorithm for finding minimum and maximum number of a given set.
5. Finding if a number is prime number or not.
6. Explain flow chart with symbols
7. Explain various basic data types available in C.
8. Explain the concept of type casting or type conversion in C.
9. Explain various storage classes available in c.
10. Explain repetitive control structures in c or loops in c with syntax and example and flowchart (for, while, do while).
11. Explain conditional statements in c
  - a. If
  - b. If else
  - c. Nested if
  - d. Else if in c with syntax, flowchart and example
12. Explain c program structure
13. Explain operators in c with example program all
14. Explain switch case with example program.
15. Explain unconditional statements in C
  - a. Goto
  - b. break

- 
- c. Continue
16. Explain command line arguments
  17. Explain printf(), scanf().
  18. Explain errors in c (types)
  19. Explain pseudo code with example
  20. What is an array and explain various kinds of arrays with syntax and example.
  21. Define array 1d, declaration, initialisation, accessing and program.
  22. Define array 2d, declaration, initialisation, accessing and program.
  23. Matrix addition, matrix multiplication, transpose of matrix program.
  24. Write a c program for sum of 10 elements in a given array.
  25. What is string? Syntax, declaration, initialisation and example.
  26. String handling functions in c.
  27. Array of strings
  28. Define structures, initializing structure and accessing with example program (tagname, typedef)
  29. Structure for student name, roll no, marks, dob.
  30. Define union with program
  31. Difference between structure and union
  32. Explain enum syntax with program
  33. Array of structures
  34. What is pointer, declaration, initialisation, accessing pointer with program.
  35. Explain pointer and arrays with program.
  36. Explain pointer to structure with program.
  37. Use of pointers in self referential structure.
  38. Explain various preprocessor directives with syntax and explain each with one program and what is preprocessor directive.
    - a. Difference between text file and binary file.
  39. Explain syntax and example program
    - a. Fgetc()
    - b. Fputc()
    - c. Fgete()
    - d. Fputs()
    - e. Fprintf()
    - f. Fscanf()
    - g. Fread()
    - h. Fwrite()
    - i. Fgete()
    - j. Fputw()
  40. Random access files with syntax and program

- 
41. Copy one file to another file
  42. Upper case or lower case.
  43. Merge 2 files
    - a. Even odd files
  44. What is a function? Explain function declaration, function header, function call, function body (definition) with example program.
  45. Explain user defined functions (4 types) with function.
  46. Explain passing parameter to functions (call by value, call by ref)
  47. Passing array to functions.
  48. Standard and library functions in c.
  49. What is recursion and factorial example
  50. Fibonacci recursion program
  51. Limitation of recursive functions
  52. Dynamic memory allocation techniques
  53. Allocating memory for array of different data types.
  54. Linear search with program
  55. Binary search example with program, complexity
  56. Bubble sort example with program (best case, average case, worst case)
  57. Selection sort example with program (best case, average case, worst case)
  58. Insertion sort example with program (best case, average case, worst case)

## 1. Creating, compiling and executing a program.

**Ans:**

Creating and running programs:

It is the job of a programmer to write and test the program. The following are four steps for creating and running programs:

- Writing and Editing the Program.
- Compiling the Program.
- Linking the Program with the required library modules.
- Executing the Program.

A. Writing and Editing Program: The Software to write programs is known as text editor. A text editor helps us enter, change and store character data. After the program is completed the program is saved in a file to disk. This file will be input to the compiler, it is known as source file. The following figure shows the various steps in building a C – program

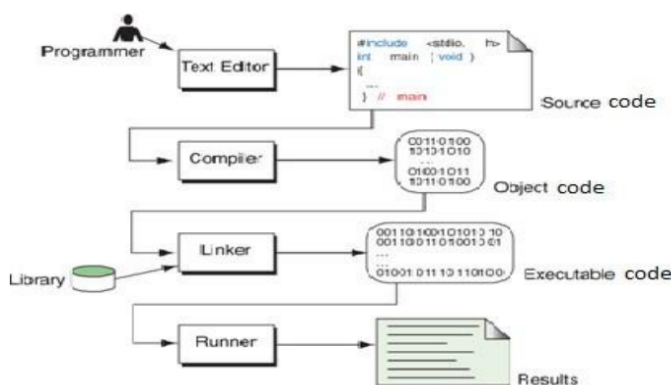


Fig: Building a C - program

- B. Compiling Programs: The code in a source file on the disk must be translated into machine language. This is the job of compiler which translates code in source file stored on disk into machine language. The C compiler is actually two separate programs. Both are combined to form the final program. An object module is the code in machine language. This module is not ready for execution because it does not have the required C and other functions included.
- C. Linking Programs: C programs are made up of many functions. Example: `printf( )` , `cos( )`... etc Their codes exists elsewhere , they must be attached to our program. The linker assembles all these functions, ours and the system's, into a final executable program.
- D. Executing Programs : Once our program has been linked, it is ready for execution. To execute a program, we use operating system command, such as `run` to load the program in to main memory and execute it. After the program processes the data, it prepares output. Data output can be to user's monitor or to a file. When program has executed, Operating System removes the program from memory.

## 2. What is algorithm, characteristics of algorithm and example.

---

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

The ordered set of instructions required to solve a problem is known as an algorithm.

The characteristics of a good algorithm are:

- **Precision** – the steps are precisely stated (defined).
- **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Finiteness** – the algorithm stops after a finite number of instructions are executed.
- **Input** – the algorithm receives input.
- **Output** – the algorithm produces output.
- **Generality** – the algorithm applies to a set of inputs.

#### **Advantages of Algorithms:**

- 1. It is a stepwise representation of a solution to a given problem, which makes it easy to understand.
- 2. An algorithm uses a definite procedure.
- 3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- 4. Every step in an algorithm has its own logical sequence so it is easy to debug.
- 5. By using algorithm, the problem is broken down into smaller pieces or steps hence; it is easier for programmer to convert it into an actual program.

#### **Disadvantages of Algorithms**

- 1. Algorithms are time consuming.
- 2. Difficult to show branching and looping in algorithms.
- 3. Big tasks are difficult to put in algorithms.

#### **Example Write an algorithm to find out number is odd or even**

Ans.

```
step 1 : start
step 2 : input number
step 3 : rem=number mod 2
step 4 : if rem=0 then
    print "number even"
        and the result of the preceding steps.
    Else
        print "number odd"
    endif
step 5 : stop
```

### **3. Algorithm for finding roots of quadratic equations.**

---

**Write an algorithm to find all roots of a quadratic equation  $ax^2+bx+c=0$ .**

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant  $D \leftarrow b^2 - 4ac$

Step 4: If  $D \geq 0$

$r1 \leftarrow (-b + \sqrt{D})/2a$

$r2 \leftarrow (-b - \sqrt{D})/2a$

    Display r1 and r2 as roots.

Else

    Calculate real part and imaginary part

$rp \leftarrow -b/2a$

$ip \leftarrow \sqrt{-D}/2a$

    Display  $rp + j(ip)$  and  $rp - j(ip)$  as roots

Step 5: Stop

○

**4. Algorithm for finding minimum and maximum number of a given set.**

- Step 1: Start
- Step 2: Declare array, min, max
- Step 3: set min = max = array[0];
- Step 4: for loop
  - If (array[i] > max)
    - Max = array[i];
  - If (array[i] < min)
    - Min = array[i];
  - End for
- Step 5: display min, max
- Step 6: stop

**5. Finding if a number is a prime number or not.**

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

```
#include <conio.h>
```

```
int main()
```

```
{
```

```
int n, i, count;
```

```
printf("Enter a number: ");
```

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

```
count = 0;
```

```
for (i = 1; i <= n; i++)
```

```
{
```

```
if (n % i == 0)
```

```
{
```



```









    count++;
}
}

if (count == 2)
    printf("%d is Prime.", n);
else
    printf("%d is not prime.", n);

getch();
return 0;
}

```

## 6. Explain flow chart with symbols

Symbol	Purpose	Description
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for arithmetic operations and data-Manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flow line
	Off-page Connector	Used to connect flowchart portion on different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

## 7. Explain various basic data types available in C.

- Basic Types:
 

They are arithmetic types and are further classified into:

  - integer types and
  - floating-point types.

### Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges:

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator. The expressions sizeof(type) yields the storage size of the object or type in bytes.

**Given below is an example to get the size of int type on any machine:**

```
#include <stdio.h>
#include <limits.h>
int main()
{
printf("Storage size for int : %d \n", sizeof(int));
return 0;
}
```

**When you compile and execute the above program, it produces the following result on Linux:**

Storage size for int : 4

### Floating-Point Types

The following table provides the details of standard floating-point types with storage sizes and value ranges and their precision:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file `float.h` defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. **The following example prints the storage space taken by a float type and its range values:**

```
#include <stdio.h>
#include <float.h>
int main()
{
    printf("Storage size for float : %d \n", sizeof(float));
    printf("Minimum float positive value: %E\n", FLT_MIN );
    printf("Maximum float positive value: %E\n", FLT_MAX );
    printf("Precision value: %d\n", FLT_DIG );
    return 0;
}
```

#### Output:

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

## 8. Explain the concept of type casting or type conversion in C.

### TYPE CONVERSION IN EXPRESSIONS

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion.

**Type conversion in c can be classified into the following two types:**

#### Implicit Type Conversion

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as implicit type conversion or type promotion. The compiler converts all operands into the data type of the largest operand.

---

The sequence of rules that are applied while evaluating expressions are given below:

- All short and char are automatically converted to int, then, If either of the operand is of type long double, then others will be converted to long double and result will be long double.
- Else, if either of the operand is double, then others are converted to double.
- Else, if either of the operand is float, then others are converted to float.
- Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.
- Else, if one of the operand is long int, and the other is unsigned int, then
- if a long int can represent all values of an unsigned int, the unsigned int is converted to long int. otherwise, both operands are converted to unsigned long int.
- Else, if either operand is long int then other will be converted to long int.
- Else, if either operand is unsigned int then others will be converted to unsigned int.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

### Explicit Type Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.

The explicit type conversion is also known as type casting. Type casting in c is done in the following form:

**(data\_type)expression;**

where, data\_type is any valid c data type, and expression may be constant, variable or expression

**For example, x=(int)a+b\*d;**

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

All integer types to be converted to float.

All float types to be converted to double.

All character types to be converted to integer

## 9. Explain various storage classes available in c.

Storage Classes:

Every Variable in a program has memory associated with it. Memory Requirement of Variables is different for different types of variables. In C, Memory is allocated & released at different place

---

### Different Storage Classes:

Auto Storage Class

Static Storage Class

Extern Storage Class

Register Storage Class

Automatic (Auto) storage class

This is default storage class All variables declared are of type Auto by default In order to Explicit declaration of variable use 'auto' keyword `auto int num1 ; // Explicit Declaration`

External ( extern ) storage class in C Programming Variables of this storage class are "Global variables" Global Variables are declared outside the function and are accessible to all functions in the program.

External ( extern ) storage class

in C Programming Variables of this storage class are "Global variables" Global Variables are declared outside the function and are accessible to all functions in the program.

Register Storage Class register keyword is used to define local variable. Local variable are stored in register instead of RAM. As variable is stored in register, the Maximum size of variable = Maximum Size of Register unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.

### 10. Explain repetitive control structures in c or loops in c with syntax and example and flowchart (for, while, do while).

**Ans.**

Repetitive control structures, or loops, in C are used to execute a block of code repeatedly while a certain condition is met. There are three types of loops in C: for loop, while loop, and do-while loop.

#### For loop:

The for loop is a control structure that repeats a block of code a fixed number of times. Its syntax is as follows:

```
for (initialization; condition; increment/decrement) {  
    // code to be executed  
}
```

Here, initialization sets the initial value of the loop control variable, condition is the expression that is tested before each iteration, and increment/decrement is the expression that updates the loop control variable at the end of each iteration.

#### Example:

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

```
int main() {
```

```
int i;
```

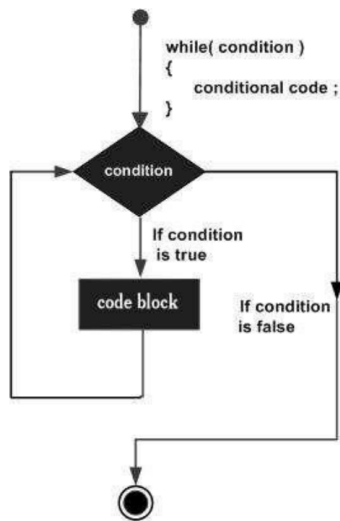
```
for(i=1; i<=5; i++) {  
    printf("%d ", i);  
}
```

```
return 0;
```

```
}
```

In this example, the loop runs 5 times and prints the values of i from 1 to 5.

#### Flowchart:



## 2. While loop:

The while loop is a control structure that repeats a block of code while a certain condition is true. Its syntax is as follows:

```
while (condition) {  
    // code to be executed  
}
```

Here, condition is the expression that is tested before each iteration.

#### Example:

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

```
int main() {  
    int i = 1;
```

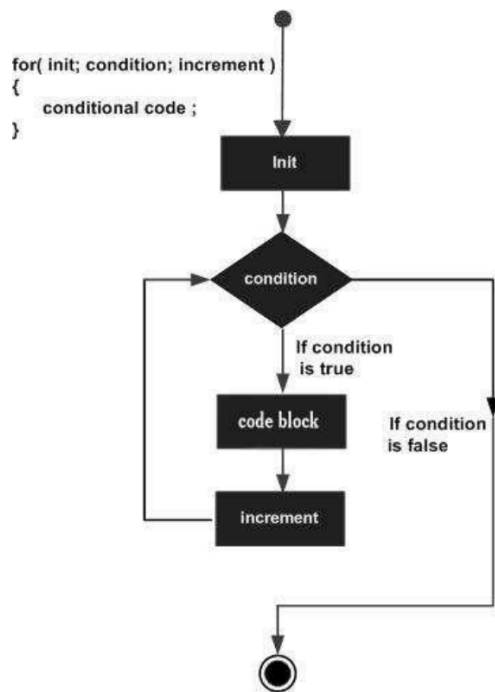
```
    while(i <= 5) {
```

```
printf("%d ", i);  
i++;  
}
```

```
return 0;  
}
```

In this example, the loop runs 5 times and prints the values of i from 1 to 5.

### Flowchart:



### 3. Do-while loop:

The do-while loop is a control structure that repeats a block of code while a certain condition is true. Its syntax is as follows:

```
do {  
    // code to be executed  
} while (condition);
```

Here, the code block is executed first, and then the condition is tested. If the condition is true, the loop runs again.

### Example:

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

```
int main() {
```

```
int i = 1;
```

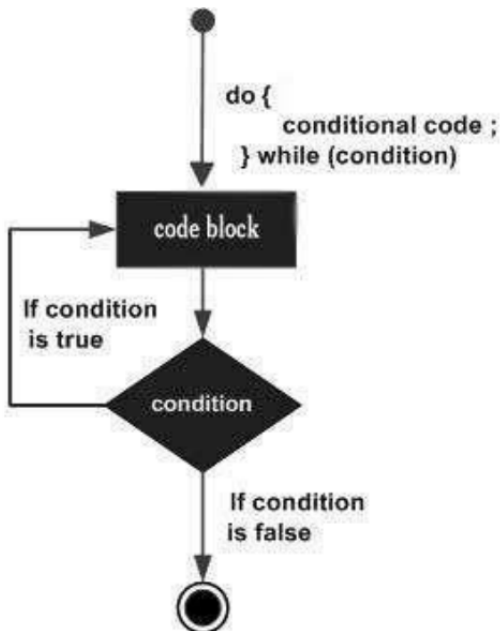
```
do {  
    printf("%d ", i);  
    i++;  
} while(i <= 5);
```

```
return 0;
```

```
}
```

In this example, the loop runs 5 times and prints the values of i from 1 to 5.

#### Flowchart:



#### 11. Explain conditional statements in c

- If
- If else
- Nested if
- Else if in c with syntax, flowchart and example

Ans.

if Statement

An if statement consists of a Boolean expression followed by one or more statements.



---

### Syntax

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

#### if...else Statement

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

### Syntax

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

#### Nested if Statements

It is always legal in C programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s)

### Syntax

The syntax for a **nested if** statement is as follows:

```
if( boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
    if(boolean_expression 2)
    {
        /* Executes when the boolean expression 2 is true */
    }
}
```

#### if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement

---

**Syntax**

The syntax of an **if...else if...else** statement in C programming language is:

```
if(boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
    /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
    /* Executes when the boolean expression 3 is true */
}
else
{
    /* executes when the none of the above condition is true */
}
```

**12. Explain c program structure**

**Ans.**

**Definition:** A structure is a collection of one or more variables of different data types, grouped together under a single name. By using structures variables, arrays, pointers etc can be grouped together.

Suppose Student record is to be stored, then for storing the record we have to group together all the information such as Roll, Name, and Percent which may be of different data types.

**13. Explain operators in c with example program all**

**Ans. pg 38**

**OPERATORS AND EXPRESSIONS**

C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

S.no	Types of Operators	Description
1	<b>Arithmetic_operators</b>	These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus
2	<b>Assignment_operators</b>	These are used to assign the values for the variables in C programs.
3	<b>Relational operators</b>	These operators are used to compare the value of two variables.
4	<b>Logical operators</b>	These operators are used to perform logical operations on the given two variables.
5	<b>Bit wise operators</b>	These operators are used to perform bit operations on given two variables.
6	<b>Conditional (ternary) operators</b>	Conditional operators return one value if condition is true and returns another value if condition is false.
7	<b>Increment/decrement operators</b>	These operators are used to either increase or decrease the value of the variable by one.
8	<b>Special operators</b>	&, *, sizeof( ) and ternary operators.

#### 14. Explain switch case with example program.

**Pg 76**

#### 15. Explain unconditional statements in C

- **Goto**
- **break**
- **Continue**

**Ans. Pg 96**

#### 16. Explain command line arguments

**Pg 66,67**

#### 17. Explain printf(), scanf().

**Ans.**

In C programming language, printf() and scanf() are two important functions used for input and output operations.

1. **printf():** It is a library function used to send formatted output to the screen. The function prints the string inside quotations. To use printf() in a program, we need to include the stdio.h header file.
2. **scanf():** It is also a library function used for taking input from the user. The scanf() function reads the input from the standard input stream stdin and scans that input according to the format provided.

Both functions are defined in stdio.h header file. The printf() function is used to print character, string, float, integer, octal and hexadecimal values onto the output screen. On the other hand, scanf() function is used to read character, string, float, integer values from standard input.

It is important to understand these functions because they are widely used in C programming language for performing input/output operations.

---

## 18. Explain errors in c (types)

**Ans.**

Errors in C programming language can be classified into five types: syntax error, runtime error, logical error, semantic error, and linker error.

1. **Syntax Error:** These errors occur when the programmer violates the grammar rules of the programming language. The compiler can easily detect these errors during compilation.
2. **Runtime Error:** These errors occur during program execution when a program tries to perform an illegal operation such as dividing by zero or accessing an invalid memory location. These errors are also called exceptions.
3. **Logical Error:** These errors occur when a program compiles and runs without any problem but produces incorrect results due to a mistake in the logic of the program.
4. **Semantic Error:** These errors occur when a program compiles and runs without any problem but produces incorrect results due to a mistake in the meaning of the code.
5. **Linker Error:** These errors occur when there is a problem with linking object files together to create an executable file. The linker can detect these errors during linking.

Syntax errors are caused by violating the rules of writing C/C++ syntax. Logical and semantic errors are mainly caused by mistakes while typing or not following the syntax of the specified programming language. On the other hand, runtime and linker errors can only be detected during program execution or linking. Understanding different types of errors in C programming language is important because it helps programmers identify and fix problems in their programs more efficiently.

## 19. Explain pseudo code with example

**Ans.**

### **PSEUDO CODE:**

Pseudocode is an informal high-level description of a computer program or algorithm. It is written in symbolic code which must be translated into a programming language before it can be executed.

### **Advantages of Pseudocode:**

1. Improves the readability of any approach.
2. It's one of the best approaches to start implementation of an algorithm.
3. Acts as a bridge between the program and the algorithm or flowchart.
4. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out.
5. In industries, the approach of documentation is essential. And that's where a pseudo-code proves vital.
6. The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

### **Disadvantages of Pseudocode:**

1. It doesn't provide visual representation of the program's logic
2. There are no accepted standards for writing pseudocodes. Designers use their own style of writing pseudocode.

---

3. Pseudo can not be compiled not executed hence its correctness cannot be verified by the computer.

**In Pseudocode, they are used to indicate common input-output and processing operations. They are written fully in uppercase.**

START: This is the start of your pseudocode.

INPUT: This is data retrieved from the user through typing or through an input device.

READ / GET: This is input used when reading data from a data file.

PRINT, DISPLAY, SHOW: This will show your output to a screen or the relevant output device.

COMPUTE, CALCULATE, DETERMINE: This is used to calculate the result of an expression.

SET, INIT: To initialize values

INCREMENT, BUMP: To increase the value of a variable

DECREMENT: To reduce the value of a variable

**20. What is an array and explain various kinds of arrays with syntax and example.**

**Ans. Pg 104,106..**

**21. Define array 1d, declaration, initialisation, accessing and program.**

**Ans. Pg 106**

**22. Define array 2d, declaration, initialisation, accessing and program.**

**Ans. Pg 111**

**23. Matrix addition, matrix multiplication, transpose of matrix program.**

**Ans.**

**Matrix addition:**

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

```
#include <conio.h>
```

```
int main()
```

```
{
```

```
int matA[3][3], matB[3][3], matC[3][3];
```

```
int i, j, p, r, q, s;
```

```
printf("Enter order of Matrix A: ");
```

```
scanf("%d %d", &p, &q);
```

```
printf("Enter order of Matrix B: ");
```

```
scanf("%d %d", &r, &s);
```

```
if (p == r && q == s)
```

```
{
```

```
printf("Enter Matrix A values: \n");
```

```
for (i = 0; i < p; i++)
```

```
for (j = 0; j < q; j++)
```

```
scanf("%d", &matA[i][j]);
```

---

```

printf("Enter Matrix B values: \n");
for (i = 0; i < r; i++)
    for (j = 0; j < s; j++)
        scanf("%d", &matB[i][j]);

printf("Addition of Matrices is: \n");
for (int i = 0; i < p; i++)
{
    for (int j = 0; j < q; j++)
    {
        matC[i][j] = matA[i][j] + matB[i][j];
        printf("%d ", matC[i][j]);
    }
    printf("\n");
}
}
else
    printf("Addition is not possible. \n");

getch();
return 0;
}

```

### Output:

```

Enter order of Matrix A: 2 2
Enter order of Matrix B: 2 2
Enter Matrix A values:
4 2
4 5
Enter Matrix B values:
2 4
5 2
Addition of Matrices is:
6 6
9 7

```

### Matrix multiplication:

```

#include <stdio.h>
#include <conio.h>

```

```

int main()
{

```

---

```
int matA[3][3], matB[3][3], matC[3][3];
int i, j, k, p, r, q, s;
printf("Enter order of Matrix A: ");
scanf("%d %d", &p, &q);
printf("Enter order of Matrix B: ");
scanf("%d %d", &r, &s);

if (p == s && r == q)
{
    printf("Enter Matrix A values: \n");
    for (i = 0; i < p; i++)
        for (j = 0; j < q; j++)
            scanf("%d", &matA[i][j]);

    printf("Enter Matrix B values: \n");
    for (i = 0; i < r; i++)
        for (j = 0; j < s; j++)
            scanf("%d", &matB[i][j]);

    printf("Multiplication of Matrices is: \n");
    for (i = 0; i < p; i++)
    {
        for (j = 0; j < p; j++)
        {
            matC[i][j] = 0;
            for (k = 0; k < q; k++)
            {
                matC[i][j] = matC[i][j] + (matA[i][k] * matB[k][j]);
            }
            printf("%d ", matC[i][j]);
        }
        printf("\n");
    }
}
else
    printf("Multiplication is not possible. \n");

getch();
return 0;
}
```

---

**Output:**

```
Enter order of Matrix A: 2 2
Enter order of Matrix B: 2 2
Enter Matrix A values:
1 4
5 1
Enter Matrix B values:
2 5
5 2
Multiplication of Matrices is:
22 13
15 27
```

**transpose of matrix:**

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

int main()
{
    int matA[5][5], matB[5][5];
    int i, j, p, r, q, s;
    printf("Enter order of Matrix A: ");
    scanf("%d %d", &p, &q);

    printf("Enter Matrix A values: \n");
    for (i = 0; i < p; i++)
        for (j = 0; j < q; j++)
            scanf("%d", &matA[i][j]);

    for (i = 0; i < q; i++)
        for (j = 0; j < p; j++)
        {
            matB[i][j] = matA[j][i];
        }

    printf("Tramspose of Matrices is: \n");
    for (i = 0; i < q; i++)
    {
        for (j = 0; j < p; j++)
        {
            printf("%d ", matB[i][j]);
        }
    }
}
```



---

```
    printf("\n");
}
getch();
return 0;
}
```

**Output:**

```
Enter order of Matrix A: 3 4
Enter Matrix A values:
4 5 6 7
3 5 6 4
5 4 3 9
Transpose of Matrices is:
4 3 5
5 5 4
6 6 3
7 4 9
```

**24. Write a c program for the sum of 10 elements in a given array.****Ans.**

```
#include <stdio.h>
void main()
{
    int n, a[15], i, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements: ", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    for (i = 0; i < n; i++)
    {
        sum += a[i];
    }

    printf("Sum = %d\n", sum);
}
```

**Output.**

---

Enter number of elements: 10  
Enter 10 elements: 5 5 3 7 1 9 8 2 4 6  
Sum = 50

**25. What is string? Syntax, declaration, initialisation and example.**

**Ans.** pg. 120

**26. String hanging functions in c.**

**Ans.** pg. 123

**27. Array of strings**

**Ans.** pg. 131

**28. Define structures, initializing structure and accessing with example program (tagname, typedef)**

**Ans.** pg. 132..

**29. Structure for student name, roll no, marks, dob.**

**Ans.**

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

```
struct student
{
    char name[15];
    int rno;
    int marks;
    float dob;
};

void main()
{
    int total;
    struct student s1 = {"Harsh", 91, 80, 5112004};
    printf("Printing student Details...");
    printf("\nName: %s \n Roll no.: %d \n Marks: %d \n dob: %.0f", s1.name, s1.rno, s1.marks, s1.dob);
}
```

**Output:**

```
Printing student Details...
Name: Harsh
Roll no.: 91
Marks: 80
dob: 5112004
```

### 30. Define union with a program.

**Ans.**

A union is one of the derived data types. Union is a collection of variables referred under a single name. The syntax, declaration and use of union is similar to the structure but its functionality is different. The general format or syntax of a union definition is as follows,

**Syntax:**

```
union union_name
{
    element 1;
    element 2;
    .....
} union_variable;
```

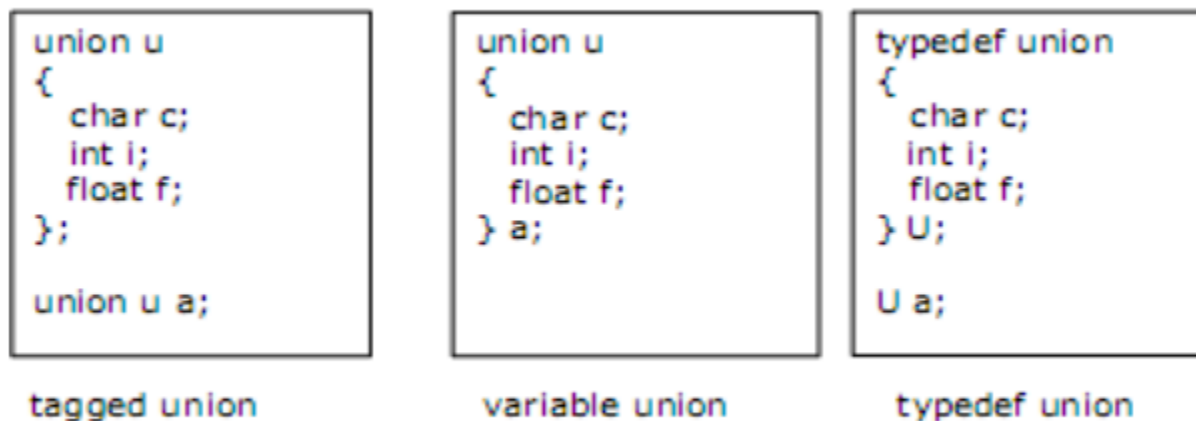
Example:

```
union techno
{
    int comp_id;
    char nm;
    float sal;
} tch;
```

A union variable can be declared in the same way as structure variable.

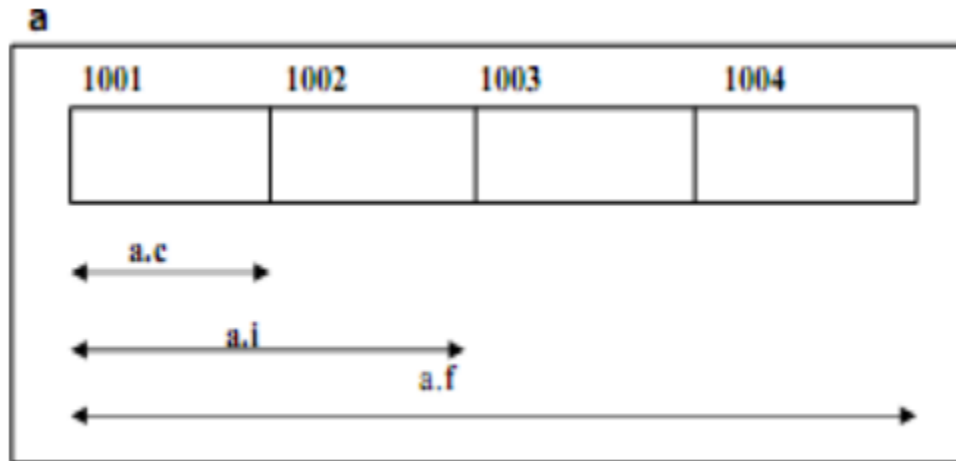
```
union tag_name var1, var2...;
```

A union definition and variable declaration can be done by using any one of the following



**Figure 5.7 Types of Union Definitions**

We can access various members of the union as mentioned: a.c a.i a.f and memory organization is shown below, In the above declaration, the member f requires 4 bytes which is the largest among all the members.



**Figure 5.8: Memory Organization Union**

Figure 5.8 shows how all the three variables share the same address. The size of the union here is 4 bytes. A union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous members' value.

### 31. Difference between structure and union.

**Ans.**

	Structure	Union
Keyword	Struct	Unio
Definition	A structure is a collection of logically related elements, possibly of different types, having a single name.	A union is a collection of logically related elements, possibly of different types, having a single name, shares single memory location.
Declaration	<pre>struct tag_name{     type1 member1; type1     member2; ..... }; struct tag_name var;</pre>	<pre>union tag_name{     type1 member1; type1     member2; ..... }; union tag_name var;</pre>
Initialization	Same	Same
Accessing	Accessed by specifying structure_ variable_name.member_name	Accessed by specifying union_ variable_name.member_name
)Memory Allocation	Each member of the structure occupies unique location, stored in contiguous locations.	Memory is allocated by considering the size of the largest member. All the members share the common location.

Size	Size of the structure depends on the type of members, adding size of all the members. <code>sizeof (st_var);</code>	Size is given by the size of the largest member <code>sizeof(un_variable)</code>
Using pointers	Structure members can be accessed by using dereferencing operator dot and selection operator (->)	same as structure
	We can have arrays as a member of structures. All members can be accessed at a time.	We can have array as a member of union. Only one member can be accessed at a time.
	Nesting of structures is possible.	same.
	It is possible structure may contain union as a member.	It is possible union may contain structure as a member

### 32. Explain enum syntax with program

**Ans.**

In C programming, an enumeration type (also called enum) is a data type that consists of integral constants. To define enums, the enum keyword is used.

```
enum flag {const1, const2, ..., constN};
```

By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary).

// Changing default values of enum constants

```
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,};
```

Enumerated Type Declaration

When you define an enum type, the blueprint for the variable is created. Here's how you can create variables of enum types.

```
enum boolean {false, true};
enum boolean check; // declaring an enum variable
```

Here, a variable check of the type enum boolean is created. You can also declare enum variables like this.

```
enum boolean {false, true} check;
```

Here, the value of false is equal to 0 and the value of true is equal to 1.

Example: Enumeration Type

```
#include enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
int main()
```

---

```
{  
    // creating today variable of enum week type  
    enum week today;  
    today = Wednesday;  
    printf("Day %d",today+1); return 0;  
}
```

Output: **Day 4**

### 33. Array of structures

**Ans.**

An array is a collection of elements of same data type that are stored in contiguous memory locations. A structure is a collection of members of different data types stored in contiguous memory locations. An array of structures is an array in which each element is a structure. This concept is very helpful in representing multiple records of a file, where each record is a collection of dissimilar data items.

As we have an array of integers, we can have an array of structures also. For example, suppose we want to store the information of class of students, consisting of name, roll\_number and marks, A better approach would be to use an array of structures. Array of structures can be declared as follows,

```
struct tag_name arrayofstructure[size];
```

Let's take an example, to store the information of 3 students, we can have the following structure definition and declaration,

```
struct student  
{  
    char name[10];  
    int rno;  
    float avg;  
};  
struct student s[3];
```

Defines an array called s, which contains three elements. Each element is defined to be of type struct student.

For the student details, array of structures can be initialized as follows,

```
struct student s[3]= {{"ABC",1,56.7},{"xyz",2,65.8},{"pqr",3,82.4}};
```

**Ex:** An array of structures for structure employee can be declared as struct employee emp[5];

Let's take an example, to store the information of 5 employees, we can have the following structure definition and declaration,

```
struct employee
```

---

```
{
```

```
int empid;
```

```
    char name[10]; 32 float salary;
```

```
};
```

```
struct employee emp[5];
```

Defines array called emp, which contains five elements. Each element is defined to be of type struct employee. For the employee details, array of structures can be initialized as follows,

```
struct employee emp[5] = {  
    {1,"ramu",25,20000},  
    {2,"ravi",65000},  
    {3,"tarun",82000},  
    {4,"rupa",5000},  
    {5,"deepa",27000}  
};
```

### 34. What is pointer, declaration, initialisation, accessing pointer with program.

**Ans.**

Pointer is a user defined data type that creates special types of variables which can hold the address of primitive data type like char, int, float, double or user defined data type like function, pointer etc. or derived data type like array, structure, union, enum.

Examples:

```
int *ptr;
```

In c programming every variable keeps two types of value.

- Value of variable.
- Address of variable where it has stored in the memory.

In C, every variable must be declared for its type. Since pointer variable contain addresses that belong to a separate data type ,they must be declared as pointers before we use them.

#### A. Declaration of a pointer variable:

The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

This tells the compiler three things about the variable pt\_name:

- a. The \* tells that the variable pt\_name is a pointer variable
- b. pt\_name needs a memory location
- c. pt\_name points to a variable of type data\_type Ex: int \*p;

Declares the variable p as a pointer variable that points to an integer data type.

---

## B. Initialization of pointer variables:

The process of assigning the address of a variable to a pointer variable is known as **initialization**. Once a pointer variable has been declared we can use assignment operator to initialize the variable.

**Ex:**

```
int quantity;
```

```
int *p;           //declaration
```

```
p=&quantity;      //initialization
```

We can also combine the initialization with the declaration:

```
int *p=&quantity;
```

Always ensure that a pointer variable points to the corresponding type of data.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. `int x, *p=&x;`

## 35. Explain pointer and arrays with programs.

**Ans.**

Pointers and Arrays :-

When an array is declared the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

**Ex:-** `static int x[5]= {1,2,3,4,5};`

Suppose the base address of x is 1000 and assuming that each integer requires two bytes. The five elements will be stored as follows.

elements	x [0]	x[1]	x[2]	x[3]	x[4]
value	1	2	3	4	5
address	1000	1002	1004	1006	1008

the name x is defined as a constant pointer pointing to the first element, x[0], and therefore the value x is 1000, the location where x[0] is stored.



---

That is  $x = \&x[0] = 1000$ ;

If we declare  $p$  as an integer pointer, then we can make the pointer  $p$  to the array  $x$  by the following assignment

$p = x$ ;            which is equivalent to  $p = \&x[0]$ ;

Now we can access every value of  $x$  using  $p++$  to move from one element to another. The relationship between  $p$  and  $x$  is shown below

$p+0 = \&x[0] = 1000$

$p+1 = \&x[1] = 1002$

$p+2 = \&x[2] = 1004$

$p+3 = \&x[3] = 1006$

$p+4 = \&x[4] = 1008$

**Note:-** address of an element in an array is calculated by its index and scale factor of the datatype

Address of  $x[n] = \text{base address} + (n * \text{scale factor of type of } x)$ .

Eg:- `int x[5]; x=1000;`

Address of  $x[3] = \text{base address of } x + (3 * \text{scale factor of int})$

$= 1000 + (3 * 2)$

$= 1000 + 6$

$= 1006$

Exg:- `float avg[20];`

`avg=2000;`

Address of  $\text{avg}[6] = 2000 + (6 * \text{scale factor of float})$

$= 2000 + 6 * 4$

$= 2000 + 24$

$= 2024$ .

Ex:- `char str [20];`

`str =2050;`

Address of  $\text{str}[10] = 2050 + (10 * 1) = 2050 + 10 = 2060$ .

Note2:- when handling arrays, of using array indexing we can use pointers to access elements.

Like  $*(p+3)$  given the value of  $x[3]$

The pointer accessing method is faster than the array indexing.

Accessing elements of an array:-

*/\*Program to access elements of a one dimensional array\*/*

*#include<stdio.h>*

```

void main()
{
int arr[5]={10,20,30,40,50};
int p=0;
printf("\n value@ arr[i] is arr[p] | *(arr+p) | *(p+arr) | p[arr] | located @address \n");
for(p=0;p<5;p++)
    { printf("\n value of arr[%d] is:",p);
      printf(" %d | ",arr[p]);
      printf(" %d | ",*(arr+p)); printf(" %d | ",*(p+arr));
      printf(" %d | ",p[arr]); printf("address of arr[%d]=%u\n",p,arr[p]);
    }
}

```

### output:

```

value@ arr[i] is arr[p] | *(arr+p) | *(p+arr) | p[arr] | located @address
value of arr[0] is: 10 | 10 | 10 | 10 | address of arr[0]=10
value of arr[1] is: 20 | 20 | 20 | 20 | address of arr[1]=20
value of arr[2] is: 30 | 30 | 30 | 30 | address of arr[2]=30
value of arr[3] is: 40 | 40 | 40 | 40 | address of arr[3]=40
value of arr[4] is: 50 | 50 | 50 | 50 | address of arr[4]=50
-

```

### 36. Explain pointer to structure with program.

Ans.

### 37. Use of pointers in self referential structure.

Ans.

#### Self-referential structure

A structure definition which includes at least one member as a pointer to the same structure is known as self-referential structure. It can be linked together to form useful data structures such as lists, queues, stacks and trees. It is terminated with a NULL pointer .

The syntax for using the self referential structure is as follows,

struct tag\_name

```

{
    Type1 member1;
    Type2 member2;
    .....
}

```

```
struct tag_name *next;
```

```
};
```

Ex:-

```
struct node
```

```
{
```

```
int data;
```

```
struct node *next;
```

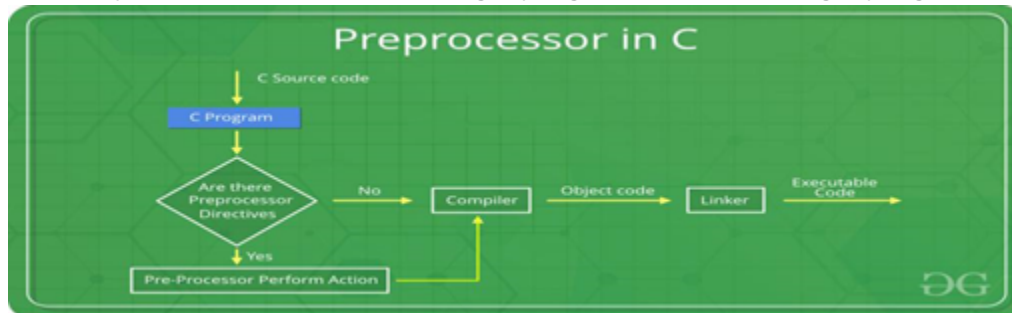
```
} n1, n2;
```

**38. Explain various preprocessor directives with syntax and explain each with one program and what is preprocessor directive. Difference between text file and binary file.**

**Ans.**

C Preprocessors:

As the name suggests Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C .



You can see the intermediate steps in the above diagram. The source code written by programmers is stored in the file program.c. This file is then processed by preprocessors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program .obj. Finally, the linker links this object code file to the object code of the library functions to generate the executable file program.exe.

- Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling.
- All of these preprocessor directives begin with a „#" (hash) symbol.
- This („#" symbol at the beginning of a statement in a C program indicates that it is a pre-processor directive. We can place these preprocessor directives anywhere in our program

There are 3 main types of preprocessor directives:

1. File Inclusion
2. Macros
3. Conditional Compilation

- 
1. File Inclusion: This type of pre-processor directive tells the compiler to include a file in the source code program. There are two types of files which can be included by the user in the program:
    - a) Header File or Standard Functions: These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O functions are in „stdio.h“ file whereas functions which perform string operations are in „string“ file.

**Syntax:**

```
#include <file_name >
```

where *file\_name* is the name of file to be included. The „<„ and „>“ brackets tells the compiler to look for the file in standard directory.

- b) User defined header files: When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:

```
#include "filename"
```

2. Macros: Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The „#define“ directive is used to define a macro. Let us now understand the macro definition with the help of a **program**:

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

```
// macro definition #define LIMIT 5 int main()
```

```
{
```

```
for (int i = 0; i < LIMIT; i++)
```

```
{
```

```
printf("%d \n",i);
```

```
}return 0;
```

```
}
```

In the above program, when the compiler executes the word LIMIT it replaces it with 5. The word „LIMIT“ in the macro definition is called a macro template and „5“ is macro expansion. Note: There is no semi-colon(„;“) at the end of macro definition. Macro definitions do not need a semi-colon to end.

Macros with arguments: We can also pass arguments to macros. Macros defined with arguments works similarly as functions. Let us understand this with a program:

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

---

```
// macro with parameter

#define AREA(l, b) (l * b)

int main()

{

int l1 = 10, l2 = 5, area; area = AREA(l1, l2);

printf("Area of rectangle is: %d", area); return 0;

}
```

We can see from the above program that whenever the compiler finds AREA(l, b) in the program it replaces it with the statement (l\*b) . Not only this, the values passed to the macro template AREA(l, b) will also be replaced in the statement (l\*b). Therefore AREA(10, 5) will be equal to 10\*5.

3. Conditional Compilation: Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions.

**39. Explain syntax and example program**

- **Fgetc()**
- **Fputc()**
- **Fgete()**
- **Fputs()**
- **Fprintf()**
- **Fscanf()**
- **Fread()**
- **Fwrite()**
- **Fgete()**
- **Fputw()**

Ans.

FUNCTION NAME	OPERATION
<b>fopen()</b>	<b>Creates a new file for use if file not exists Opens an existing file for use</b>
<b>fclose()</b>	<b>Closes a file which has been opened for use</b>
<b>fcloseall()</b>	<b>Closes all files which are opened</b>
<b>getc()/fgetc()</b>	<b>Reads a character from a file</b>

<b>putc()/fputc()</b>	<b>Writes a character to a file</b>
<b>fprintf()</b>	<b>Writes a set of data values to files</b>
<b>fscanf()</b>	<b>Reads a set of data values from files</b>
<b>getw()</b>	<b>Reads an integer from file</b>
<b>putw()</b>	<b>Writes an integer to a file</b>
<b>gets()</b>	<b>Reads a string from a file</b>
<b>puts()</b>	<b>Writes a string to a file</b>
<b>fseek()</b>	<b>Sets the position to a desired point in afile</b>
<b>ftell()</b>	<b>Gives the current position in the file</b>
<b>rewind()</b>	<b>Sets the position to the beginning of the file</b>

### **Naming and opening a file:**

A name is given to the file used to store data. The file name is a string of characters that make up a valid file name for operating system. It contains two parts. A primary name and an optional period with the extension.

**Examples: Student.dat, file1.txt, marks.doc, palin.c.**

**The general format of declaring and opening a file is**

```
FILE *fp;      //declaration
fp=fopen ("filename","mode");    //statement to open file.
```

Here FILE is a data structure defined for files. fp is a pointer to data type FILE. filename is the name of the file. mode tells the purpose of opening this file.

**Reading data from file :** Input functions used are ( Input operations on files)

#### **a)getc();**

It is used to read characters from file that has been opened for read operation. Syntax: c=getc (file pointer)

This statement reads a character from file pointed to by file pointer and assign to c. It returns an end-of-file marker EOF, when end of file has been reached

#### **b)fscanf();**

This function is similar to that of scanf function except that it works on files.

**Syntax:***fscanf (fp, "control string", list);*

**Example** `fscanf(f1,"%s%d",str,&num);`

---

The above statement reads string type data and integer type data from file.

**c) getw();**

This function reads an integer from file. Syntax: getw (file pointer);

**d) fgets();**

This function reads a string from a file pointed by a file pointer. It also copies the string to a memory location referred by an array.

**Syntax: fgets(string,no of bytes,filepointer);**

**e) fread();**

This function is used for reading an entire structure block from a given file. Syntax: fread(&struct\_name,sizeof(struct\_name),1,filepointer);

**Writing data to a file:**

To write into a file, following C functions are used

**a. putc();**

This function writes a character to a file that has been opened in write mode.

**Syntax: putc(c,fp);**

This statement writes the character contained in character variable c into a file whose pointer is fp.

**b. fprintf();** This function performs function, similar to that of printf. Syntax: fprintf(f1,"%s,%d",str,num);

**c. putw();** It writes an integer to a file.

**d. fputs();** This function writes a string into a file pointed by a file pointer.

**Syntax: fputs(string, filepointer);**

**e. fwrite();** This function is used for writing an entire block structure to a given file. Syntax: fwrite(&struct\_name, sizeof(struct\_name),1,filepointer);

**Closing a file:**

A file is closed as soon as all operations on it have been completed. Closing a file ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. Another instance where we have to close a file is to reopen the same file in a different mode. Library function for closing a file is

**Syntax: fclose(file pointer);**

**Example: fclose(fp);**

---

Where fp is the file pointer returned by the call to fopen(). fclose() returns 0 on success (or) -1 on error. Once a file is closed, its file pointer can be reused for another file. Note: fcloseall() can be used to close all the opened files at once.

**40. Random access files with syntax and program.**

**Ans.**

At times we needed to access only a particular part of a file rather than accessing all the data sequentially, which can be achieved with the help of functions fseek, ftell and rewind available in IO library.

**ftell():-**

ftell takes a file pointer and returns a number of type long, that corresponds to the current position. This function is useful in saving the current position of the file, which can later be used in the program.

**Syntax: n=ftell(fp);**

n would give the Relative offset (In bytes) of the current position. This means that already n bytes have a been read or written

**rewind():-**

It takes a file pointer and resets the position to the start of the file.

**Syntax: rewind(fp);**

**n=ftell(fp);**

would assign 0 to n because the file position has been set to start of the file by rewind(). The first byte in the file is numbered 0, second as 1, so on. This function helps in reading the file more than once, without having to close and open the file.

Whenever a file is opened for reading or writing a rewind is done implicitly.

**fseek:-**

fseek function is used to move the file pointer to a desired location within the file.

**Syntax: fseek(file ptr,offset,position);**

file pointer is a pointer to the file concerned, offset is a number or variable of type long and position is an integer number which takes one of the following values. The offset specifies the number of positions(Bytes) to be moved from the location specified by the position which can be positive implies moving forward and negative implies moving backwards.

POSITION VALUE	VALUE CONSTANT	MEANING
0	SEEK_SET	BEGINNING OF FILE
1	SEEK_CUR	CURRENT POSITION
2	SEEK_END	END OF FILE



---

**Example: fseek(fp,10,0) ;**

**fseek(fp,10,SEEK\_SET);**// file pointer is repositioned in the forward direction 10 bytes.

**fseek(fp,-10,SEEK\_END);** // reads from backward direction from the end of file.

When the operation is successful fseek returns 0 and when we attempt to move a file beyond boundaries fseek returns -1.

Some of the Operations of fseek function are as follows:

#### **Program on random access to files:**

```
#include<stdio.h>
void main()
{
    FILE *fp;
    char ch;
    fp=fopen("my1.txt","r");
    fseek(fp,21,SEEK_SET);
    ch=fgetc(fp);
    while(!feof(fp))
    {
        printf("%c\t",ch);
        printf("%d\n",ftell(fp));
        ch=fgetc(fp);
    }
    rewind(fp);
    printf("%d\n",ftell(fp));
    fclose(fp);
}
```

#### **41. Copy one file to another file**

**Ans.**

Program:

```
#include<stdio.h>
void main()
{
    FILE *f1,*f2;
```

---

```

char ch;
f1=fopen("mynew2.txt","w");
printf("\n enter some text here and press ctrl + D or ctrl + Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,f1);
fclose(f1);
f1=fopen("mynew2.txt","r");
f2=fopen("dupmynew2.txt","w");
while((ch=getc(f1))!=EOF)
    putc(ch,f2);
fcloseall();
printf("\n the copied file contents are :");
f2 = fopen("dupmynew2.txt","r");
while( ( ch = fgetc(f2) ) != EOF )
    putchar(ch);
fclose(f2);
}

```

## 42. Uppercase or lowercase.

**Ans.**

*// Write a C program which copies one file to another, replacing all lower case to UPPER case characters.*

```

#include <stdio.h>
#include <conio.h>
#include <ctype.h>

```

```

int main()
{
    char x;
    FILE *fp1, *fp2;
    fp1 = fopen("one.txt", "r");
    fp2 = fopen("two.txt", "w");
    if (fp1 == NULL)
    {
        printf("\n File doesn't exist.");
    }
    else
    {
        x = fgetc(fp1);
        while (x != EOF)
        {
            x = toupper(x);
            fputc(x, stdout);

```

```

        fputc(x, fp2);
        x = fgetc(fp1);
    }
    printf("\nContents of one file copied to another file successfully");
}
fclose(fp1);
fclose(fp2);

getch();
return 0;
}

```

#### 43. a) Merge 2 files

##### b) Even odd files

##### Ans. a)

// Write a C program to merge two files into a third file.

```

#include <stdio.h>
#include <conio.h>
#include <string.h>

```

```

int main()
{
    char x;
    FILE *fp1, *fp2, *fp3;
    fp1 = fopen("one.txt", "r");
    fp2 = fopen("two.txt", "r");
    fp3 = fopen("three.txt", "w");
    if (fp1 == NULL && fp2 == NULL)
    {
        printf("\n File doesn't exist.");
    }
    else
    {
        x = fgetc(fp1);
        while (x != EOF)
        {
            fputc(x, stdout);
            fputc(x, fp3);
            x = fgetc(fp1);
        }
        x = fgetc(fp2);
        while (x != EOF)

```

---

```

        {
            fputc(x, stdout);
            fputc(x, fp3);
            x = fgetc(fp2);
        }
        printf("\nContents of both the files copied successfully.");
    }
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);

    getch();
    return 0;
}

```

**Ans. b)**

```

/*Printing odd numbers in odd file and even numbers in even file*/
#include<stdio.h>
void main()
{
    int x,i;
    FILE *f1,*fo,*fe; //creating a file pointer
    f1=fopen("num.txt","r");//open anil in read mode to read data
    fo=fopen("odd.txt","w");
    fe=fopen("even.txt","w");
    while((x=getw(f1))!=EOF)
    {
        printf("%d\t",x);
        if(x%2==0)
            putw(x,fe);
        else
            putw(x,fo);
    }
    fcloseall();

    fo=fopen("odd.txt","r");
    printf("\n contents of odd file are :\n");
    while((x=getw(fo))!= EOF)
        printf(" %d\t",x);

    fe=fopen("even.txt","r");
    printf("\n contents of even file are :\n");
}

```

---

```
while((x=getw(fe)) != EOF)
    printf(" %d\t",x);
fcloseall();
}
```

**44. What is a function? Explain function declaration, function header, function call, function body (definition) with example program.**

**Ans.**

**Function in C:**

A function is a block of code that performs a specific task. It has a name and it is reusable. It can be executed from as many different parts in a program as required, it can also return a value to calling program.

All executable code resides within a function. It takes input, does something with it, then give the answer. A C program consists of one or more functions.

A computer program cannot handle all the tasks by itself. It requests other program like entities called functions in C. We pass information to the function called arguments which specified when the function is called. A function either can return a value or returns nothing. Function is a subprogram that helps reduce coding.

A user can create their own functions for performing any specific task of program are called user defined functions. To create and use these function we have to know these 3 elements.

- Function Declaration
- Function Definition
- Function Call

**1. Function declaration:**

The program or a function that calls a function is referred to as the calling program or calling function. The calling program should declare any function that is to be used later in the program this is known as the function declaration or function prototype.

**2. Function Definition:**

The function definition consists of the whole description and code of a function. It tells that what the function is doing and what are the input outputs for that. A function is called by simply writing the name of the function followed by the argument list inside the parenthesis. Function definitions have two parts:

- Function Header
  - The first line of code is called Function Header.
  - int sum( int x, int y)
  - It has three parts
    - The name of the function i.e. sum

- The parameters of the function enclosed in parenthesis
- Return value type i.e. int

- Function Body

Whatever is written with in { } is the body of the function.

### 3. Function Call:

In order to use the function we need to invoke it at a required place in the program. This is known as the function call.

### Simple Example of Function in C:

```
#include<stdio.h>
#include <conio.h>
int addition (int, int); //Function Declaration
int addition (int a, int b) //Function Definition
{
    int r;
    r=a + b;
    return (r);
}
int main()
{
    int z;
    z= addition(10,3); //Function Call
    printf ("The Result is %d", z);
    return 0;
}
```

**Output:** The Result is 13

45. Explain user defined functions (4 types) with function.

Ans.

A function depending on whether arguments are present or not and whether a value is returned or not may belong to any one of the following categories:

- Functions with no arguments and no return values.
- Functions with arguments and no return values.
- Functions with arguments and return values.
- Functions with no arguments and return values.

### (i) Functions with no arguments and no return values:-

When a function has no arguments, it does not return any data from calling function. When a function does not return a value, the calling function does not receive any data from the called function. That is there is no data transfer between the calling function and the called function.

---

### Example

```
#include<stdio.h>
#include<conio.h>
void add();
void main()
{
    add();
}
void add()
{
    int x=10,y=20,z;
    z=x+y;
    printf ("sum=%d",z);
}
```

**Output** : sum=30.

### (ii) Functions with arguments and no return values:-

When a function has arguments data is transferred from calling function to called function. The called function receives data from calling function and does not send back any values to calling function. Because it doesn't have return value.

#### Example

```
#include<stdio.h>
#include<conio.h>
void add(int , int);
void main()
{
    int a=10,b=20;
    add(a,b);
}
void add(int x,int y)
{
    int z;
    z=x+y;
    printf ("sum=%d",z);
}
```

**Output** : sum=30.

### (iii) Function with no arguments and return type:-

When function has no arguments data cannot be transferred to called function. But the called function can send some return value to the calling function.

#### Example

---

```
#include<stdio.h>
#include<conio.h>
int add();
void main()
{
    int sum;
    sum=add();
    printf ("sum=%d",sum);
}
int add()
{
    int x=10,y=20,z;
    z=x+y;
    return z;
}
```

**Output** : sum=30.

#### (iv) Functions with arguments and return values:-

In this data is transferred between calling and called function. That means called function receives data from calling function and called function also sends the return value to the calling function.

**Example:**

```
#include<stdio.h>
#include<conio.h>
int add(int , int);
void main()
{
    int a=10,b=20,sum;
    sum=add(a,b);
    printf ("sum=%d",sum);
}
int add(int x , int y)
{
    int z;
    z=x+y;
    return z;
}
```

**Output** : sum=30.

#### 46. Explain passing parameter to function (call by value, call by ref)

**Ans.**

Most programming languages have 2 strategies to pass parameters. They are



- 
- pass by value( Call by Value)
  - pass by reference (Call by Reference)

### (i) Pass by value (or) call by value :-

In this method calling function sends a copy of actual values to called function, but the changes in called function does not reflect the original values of calling function.

#### Example program:

```
#include<stdio.h>
void fun1(int, int);
void main( )
{
    int a=10, b=15;
    fun1(a,b);
    printf("a=%d,b=%d", a,b);
}
void fun1(int x, int y)
{
    x=x+10;
    y= y+20;
}
```

**Output:** a=10 b=15

The result clearly shown that the called function does not reflect the original values in main function.

### (ii) Pass by reference (or) call by address :-

In this method calling function sends address of actual values as a parameter to called function, called function performs its task and sends the result back to calling function. Thus, the changes in called function reflect the original values of calling function. To return multiple values from called to calling function we use pointer variables.

Calling function needs to pass „&“ operator along with actual arguments and called function need to use „\*“ operator along with formal arguments. Changing data through an address variable is known as indirect access and „\*“ is represented as indirection operator.

#### Example program:

```
#include<stdio.h>
void fun1(int,int);
void main( )
{
    int a=10, b=15;
    fun1(&a,&b);
    printf("a=%d,b=%d", a,b);
}
void fun1(int *x, int *y)
{

```

---

```
*x = *x + 10;
*y = *y + 20;
}
```

**Output:** a=20 b=35

The result clearly shown that the called function reflect the original values in main function. So that it changes original values.

#### 47. Passing array to functions.

**Ans.**

Arrays with functions:

To process arrays in a large program, we need to pass them to functions. We can pass arrays in two ways:

- 1) pass individual elements
- 2) pass the whole array.

##### Passing Individual Elements:

###### One-dimensional Arrays:

We can pass individual elements by either passing their data values or by passing their addresses. We pass data values i.e; individual array elements just like we pass any data value .As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, a variable or an expression.

###### Two-dimensional Arrays:

The individual elements of a 2-D array can be passed in the same way as the 1-D array. We can pass 2-D array elements either by value or by address.

Example:

```
#include<stdio.h>
void fun1(int a)
main()
{
    int a[2][2]={1,2,3,4};
    fun1(a[0][1]);
}
void fun1(int x)
{
    printf("%d",x+10);
}
```

##### Passing an entire array to a function:

###### One-dimensional array:

---

To pass the whole array we simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an array. We do not need to specify the number of elements.

### **Two-dimensional array:**

When we pass a 2-D array to a function, we use the array name as the actual parameter just as we did with 1-D arrays. The formal parameter in the called function header, however must indicate that the array has two dimensions.

#### **Rules:**

- The function must be called by passing only the array name.
- In the function definition, the formal parameter is a 2-D array with the size of the second dimension specified.

Example:

```
#include<stdio.h>
void fun1(int a[][2]);
void main()
{
    int a[2][2]={1,2,3,4};
    fun1(a);
}
void fun1(int x[][2])
{
    int i,j;
    for(i=0;i<2;i++)
    for(j=0;j<2;j++)
    printf("%d",x[i][j]);
}
```

## **48. Standard and library functions in c.**

### **Ans.**

STANDARD FUNCTIONS AND LIBRARY FUNCTIONS:

C functions can be classified into two categories, namely, library functions and user-defined functions. `main` is the example of user-defined functions. `printf` and `scanf` belong to the category of library functions. The main difference between these two categories is that library functions are not required to be written by us where as a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library.

ANSI C Standard Library Functions:

The C language is accompanied by a number of library functions that perform various tasks. The ANSI committee has standardized header files which contain these functions.

---

Some of the Header files are:

<math.h>	Mathematical functions
<stdio.h>	Standard I/O library functions
<stdlib.h>	Utility functions such as string conversion routines memory allocation routines , random number generator , etc.
<string.h>	String Manipulation functions
<time.h>	Time Manipulation Functions

#### MATH.H

The math library contains functions for performing common mathematical operations.

abs: Returns the absolute value of an integer x

cos: Returns the cosine of x, where x is in radians

exp: Returns "e" raised to a given power

fabs: Returns the absolute value of a float x

log: Returns the logarithm to base e

log10: Returns the logarithm to base 10

pow : Returns a given number raised to another number

sin: Returns the sine of x, where x is in radians

sqrt: Return the square root of x

tan: Return the tangent of x, where x is in radians

ceil: The ceiling function rounds a number with a decimal part up to the next highest

integer (written mathematically as  $\lceil x \rceil$ )

floor : The floor function rounds a number with a decimal part down to the next lowest integer  
(written mathematically as  $\lfloor x \rfloor$ )

#### STDIO.H

Standard I/O library functions

#### STDLIB.H

Utility functions such as string conversion routines memory allocation routines, random number generator, etc.

atof: Convert a string to a double (not a float)

atoi: Convert a string to an int

exit: Terminate a program, return an integer value

free: Release allocated memory

malloc: Allocate memory

rand: Generate a pseudo-random number

system: Execute an external command

---

## STRING.H

There are many functions for manipulating strings. Some of the more useful are:

strcat : Concatenates (i.e., adds) two strings

strcmp: Compares two strings

strcpy: Copies a string

strlen: Calculates the length of a string (not including the null)

strstr: Finds a string within another string

strtok: Divides a string into tokens (i.e., parts)

## TIME.H

This library contains several functions for getting the current date and time.

time: Get the system time

ctime: Convert the result from time() to something meaningful

## 49. What is recursion and factorial example

**Ans.**

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function. The main advantage of recursive functions is that we can use them to create clearer and simpler versions of several programs.

### Syntax:-

A function is recursive if it can call itself; either **directly**:

```
void f( )
{
    f( );
}
```

**(or) indirectly:**

```
void f( )
{
    g( );
}
void g( )
{
    f( );
}
```

---

Recursion rule 1: Every recursive method must have a base case -- a condition under which no recursive call is made -- to prevent infinite recursion.

Recursion rule 2: Every recursive method must make progress toward the base case to prevent infinite recursion

### Factorial Example:

```
#include<stdio.h>
int fact(int);
main()
{
    int n,f;
    printf("\n Enter any
    number:"); scanf("%d",&n);
    f=fact(n);
    printf("\n Factorial of %d is %d",n,f);
}
int fact(int n)
{
    int f;
    if(n==0||n==1) //base case
        f=1;
    else
        f=n*fact(n-1); //recursive case
    return f;
}
```

**Output:-** Enter any number: 5

Factorial of 5 is 120

## 50. Fibonacci recursion program

**Ans.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int f,f1,f2,n,i,res;
    printf("enter the limit:");
    scanf("%d",&n);
    printf("The fibonacci series is:");
    for(i=0;i<n;i++)
    {
        printf("%d\t", fib(i));
    }
}
```

---

```
    }  
}  
int fib(int i)  
{  
    if(i==0)  
        return 0;  
    else if(i==1)  
        return 1;  
    else  
        return(fib(i-1)+fib(i-2));  
}
```

## 51. Limitation of recursive functions

**Ans.**

1. Recursive solutions may involve extensive overhead because they use function calls.
2. Each function call requires push of return memory address, parameters, returned results, etc. and every function return requires that many pops.
3. Each time we make a call we use up some of our memory allocation. If the recursion is deep that is, if there are many recursive calls then we may run out of memory.
4. Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
5. Aborting a recursive process in midstream is slow and sometimes nasty.
6. Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
7. It is difficult to find bugs, particularly when using global variables.
8. Recursion uses more processor time.

## 52. Dynamic memory allocation techniques

Function	Use of Function
<u>malloc()</u>	Allocates requested size of bytes and returns a pointer first byte of allocated space
<u>calloc()</u>	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
<u>free()</u>	deallocate the previously allocated space
<u>realloc()</u>	Change the size of previously allocated space

### **malloc()**

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

#### **Syntax:**

**ptr=(cast-type\*)malloc(byte-size)**

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

**ptr=(int\*)malloc(100\*sizeof(int));**

### **calloc()**

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

#### **Syntax:**

**ptr=(cast-type\*)calloc(n,element-size);**

This statement will allocate contiguous space in memory for an array of elements. For example:

**ptr=(float\*)calloc(25,sizeof(float));**

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes

### **free()**

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

#### **syntax:**

**free(ptr);**

This statement causes the space in memory pointer by ptr to be deallocated.



---

### **realloc():**

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

#### **Syntax:**

**ptr=realloc(ptr,newsize);**

### **Examples of calloc(), realloc():**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    clrscr();
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    printf("Address of newly allocated memory: ");
    for(i=0;i<n1+n2;++i)
        printf("%u\t",ptr+i);
    getch();
    return 0;
}
```

### **Examples of malloc():**

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include<stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
    }
}
```

---

```
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",&ptr[i]);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

**Output:**

```
Enter number of elements: 5
Enter elements of array: 1 2 3 4 5
Sum=15
```

**53. Allocating memory for array of different data types.****Ans.**

In C language, allocating memory for arrays of different data types involves using the `malloc()` function, which dynamically allocates memory during program execution.

To allocate memory for an array of a specific data type, you need to know the size of that data type. For example, the size of an integer in C language is usually 4 bytes, the size of a float is 4 bytes, and the size of a double is 8 bytes.

Here's an example of how to allocate memory for an array of 10 integers using `malloc()` function in C:

```
int *myArray;
myArray = (int*) malloc(10 * sizeof(int));
```

The `malloc()` function takes the number of bytes to be allocated as an argument, so you need to multiply the number of elements in the array by the size of the data type (using the `sizeof` operator) to determine the number of bytes to allocate.

In the above example, the `malloc()` function allocates 40 bytes of memory for an array of 10 integers, each of which is 4 bytes in size. The `(int*)` before `malloc` indicates that the memory block allocated should be treated as an array of integers.

Similarly, to allocate memory for an array of 10 floats, you would use the following code:

```
float *myArray;
```

---

```
myArray = (float*) malloc(10 * sizeof(float));
```

This code allocates 40 bytes of memory for an array of 10 floats, each of which is 4 bytes in size.

Once you have allocated memory for an array using `malloc()`, you can access and manipulate its elements just like a regular array. However, it's important to remember to deallocate the memory when you're finished using it, using the `free()` function:

```
free(myArray);
```

This frees up the memory that was allocated for the array, making it available for other parts of the program to use.

#### 54. Linear search with program

Ans. Pg 219

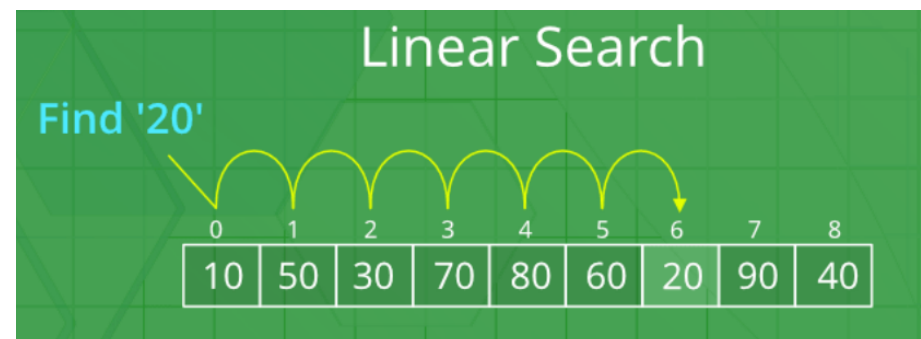
##### Linear Search

Linear search sequentially checks each element of the list until it finds one element that matches the key value. If the algorithm reaches the end of list, then it returns that the element is not found .

##### Example:

For example an array consist of 5 elements 10,50,30,70,80,60,20,90,40.

If the key value is 20.then key is compared with each element in the array until it found



##### Algorithm :

##### Linear\_Search(A, n, key)

Step 1: Repeat For I =0 to N-1

Begin

Step 2: do if key=A[i]

Step 3: then display “key is found” and go to step 5

End For

Step 4: display “key is not found”

Step 5: Exit

##### Program:

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

---

```

#include<conio.h>
void main()
{
    int ar[10],n,i,key;
    clrscr();
    printf("Enter the No.Of elements:");
    scanf("%d",&n);
    printf("Enter %d elements:",n);
    for(i=0;i<n;i++)
        scanf("%d",&ar[i]);
    printf("\nEnter Key:");
    scanf("%d",&key);
    //linear search logic
    for(i=0;i<n;i++)
        if(key==ar[i])
        {
            printf("\nThe key element is found at %d position.",i+1);
            getch();
            exit(0);
        }
    printf("\nThe element is not found.");
    getch();
}

```

### Output:

Enter the No. Of elements:9  
Enter 9 elements:10 50 30 70 80 60 20 90 40  
Enter Key:20  
The key element is found at 7 positions.

### Complexity of Linear\_Search:

The worst case complexity of linear search is  $O(n)$ .

## 55. Binary search example with program, complexity

Ans. Pg 221

### Binary Search

Binary search is used to find an element in a sorted array. If the array isn't sorted, you must sort it using a sorting technique. If the element to search is present in the list, then we print its location. The program assumes that the input numbers are in ascending order

**Example:** For example an array consist of 10 elements And key is 23 then

# Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
23 > 16 take 2 <sup>nd</sup> half	L=0	1	2	3	M=4	5	6	7	8	H=9
	2	5	8	12	16	23	38	56	72	91
23 > 56 take 1 <sup>st</sup> half	0	1	2	3	4	L=5	6	M=7	8	H=9
	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5	0	1	2	3	4	L=5, M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91

## Algorithm :

### Binary-Search [A, n, key]

Step 1: Set low to 0

Step 2: Set high to n-1

Step 3: while low <= high

Step 4: do mid =(low +high)/2

Step 5: if key=A[mid]

Step 6: then display "The key element is found" and go to step 10

Step 7: else if key >A[mid]

Step 8: then low = mid + 1

Step 9: else high = mid – 1

End While

Step 10: Display "Key not found"

Step 11: Exit

## Program:

//Bubble sort

#include<stdio.h>

#include<conio.h>

void main()

{

int ar[10],n,i,j,low,high,mid,key;

clrscr();

printf("Enter the No.Of elements:");

scanf("%d",&n);

printf("Enter %d elements:",n);

for(i=0;i<n;i++)

scanf("%d",&ar[i]);

printf("Enter Key:");

---

```

scanf("%d",&key);
//Bubble sort logic
low=0;
high=n-1;
while(low<=high)
222
{
mid=(low+high)/2;
if(key==ar[mid])
{
printf("\nThe key element is found at %d position.",mid+1);
getch();
exit(0);
}
else if(key<ar[mid])
high=mid-1;
else
low=mid+1;
}
printf("\nThe element is not found.");
getch();
}

```

### Output:

```

Enter the No.Of elements: 10
Enter %d elements: 2 5 8 12 16 23 38 56 72 91
Enter Key: 23
The key element is found at 6 position.

```

### Complexity of Binary\_Search:

The compexity of binary search is  $O(\log n)$ .

## 56. Bubble sort example with program (best case, average case, worst case)

Ans. **Pg 224**

### Bubble sort

The bubble sort is an example of exchange sort. In this method, repetitive comparison is performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming.

### Algorithm:

**Bubble\_Sort (A [ ], N)**

**Step 1:** Repeat for  $I = 0$  to  $N - 1$

---

Begin

**Step 2:** Repeat for J = 0 to N - 1 - 1

Begin

**Step 3:** If (A [J] > A [J + 1])

Swap (A [J], A [J + 1])

End For

End For

**Step 4:** Exit

**Program:**

*//Bubble sort*

*#include<stdio.h>*

*#include<conio.h>*

void main()

```
{
    int ar[10],n,i,j,temp;
    clrscr();
    printf("Enter the No.Of elements:");
    scanf("%d",&n);
    printf("Enter %d elements:",n);
    for(i=0;i<n;i++)
        scanf("%d",&ar[i]);
    //Bubble sort logic
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-1-i;j++)
        {
            if(ar[j]>ar[j+1])
            {
                temp=ar[j];
                ar[j]=ar[j+1];
                ar[j+1]=temp;
            }
        }
    }
    printf("\nThe sorted elements are:\n");
    for(i=0;i<n;i++)
        printf("%d\t",ar[i]);
    getch();
}
```

**Output:**

---

Enter the No.Of elements:5

Enter 5 elements:5 1 4 2 8

The sorted elements are:

1 2 4 5 8

### Complexity of Bubble\_Sort:

The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is

$$n(n-1)/2 \approx n^2/2$$

The worst case complexity for Bubble sort is  $O(n^2)$  and best case is  $O(n)$ .

## 57. Selection sort example with program (best case, average case, worst case)

**Ans. Pg 227**

### Selection sort:

The selection sort is nearly the same as exchange sort. Assume that we have a list on n elements. By applying selection sort, the first element is compared with all remaining (n-1) elements. The lower element is placed at the first location. Again, the second element is compared with remaining (n-1) elements. At the time of comparison, the smaller element is swapped with larger element. In this type, entire array is checked for smallest element and then swapping is done.

#### Algorithm:

#### Selection\_Sort ( A [ ], N )

Step 1 : Repeat for I = 0 to N - 2

    Begin

Step 2 : Set MIN = I

Step 3 : Repeat for J = I + 1 to N-1

    Begin

        If A[ J ] < A [ MIN ]

            Set MIN = J

    End For

Step 5 : Swap A [ I ] with A [ MIN ]

    End For

Step 6 : Exit

### Program:

//Selection sort

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int ar[10],n,i,j,min,temp;
```

```
clrscr();
```

```
printf("Enter the No.Of elements:");
```

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



---

```

printf("Enter %d elements:",n);
for(i=0;i<n;i++)
    scanf("%d",&ar[i]);
//Selection sort logic
for(i=0;i<n-1;i++)
{
    min=i;
    for(j=i+1;j<n;j++)
    {
        if(ar[j]<ar[min])
            min=j;
    }
    temp=ar[i];
    ar[i]=ar[min];
    ar[min]=temp;
}
printf("\nThe sorted elements are:\n");
for(i=0;i<n;i++)
    printf("%d\t",ar[i]);
getch();
}

```

#### Output:

```

Enter the No.Of elements:5
Enter 5 elements:9 4 7 5 8
The sorted elements are:
4 5 7 8 9

```

#### Complexity of Selection\_Sort

selection sort is a sorting algorithm, specifically an in-place comparison sort. It has  $O(n^2)$  time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.

#### 58. Insertion sort example with program (best case, average case, worst case)

**Ans. Pg 229**

##### Insertion sort

In insertion sort the element is inserted at an appropriate place. For example, consider an array of  $n$  elements. In this type also swapping of elements is done without taking any temporary variable. The greater numbers are shifted towards end locations of the array and smaller are shifted at beginning of the array.

---

**Algorithm:****Insertion\_Sort ( A [ ], N )**

```
Step 1 : Repeat For I = 1 to N - 1
    Begin
Step 2 :   Set Temp = A [ I ]
Step 4 :   Repeat For J=I to J >= 0 AND A [ J-1 ] > Temp
        Begin
            Set A [J] = A [J-1]
            Set J = J - 1
        End For
Step 5 :   Set A [ J ] = Temp
        End For
Step 4 : Exit
```

**Program:**

```
//Insertion sort
#include<stdio.h>
#include<conio.h>
void main()
{
    int ar[10],n,i,j,temp;
    clrscr();
    printf("Enter the No.Of elements:");
    scanf("%d",&n);
    printf("Enter %d elements:",n);
    for(i=0;i<n;i++)
        scanf("%d",&ar[i]);
    //Insertion sort logic
    for(i=1;i<n;i++)
    {
        temp=ar[i];
        for(j=i;j>0 && ar[j-1]>temp;j--)
        {
            ar[j]=ar[j-1];
        }
        ar[j]=temp;
    }
    printf("\nThe sorted elements are:\n");
    for(i=0;i<n;i++)
        printf("%d\t",ar[i]);
    getch();
}
```

---

**Output:**

Enter the No.Of elements:5

Enter 5 elements: 9 4 7 5 8

The sorted elements are:

4 5 7 8 9

**Complexity of Insertion Sort**

Best Case:  $O(n)$

Average Case:  $O(n^2)$

Worst Case:  $O(n^2)$