FOP WINTER PAPER SOLUTION

Q.1 (a)

I. Explain the scanf() and printf() function.

printf() function:

- printf() is used to **display output** on the screen.
- It is defined in the **stdio.h** header file.
- It can print text (strings), variables, numbers, etc.
- General syntax:

  printf("format specifier or message", variable_name);

- Example:

  int a = 10;

  printf("Value of a = %d", a);

- Here:
  - "Value of a = %d" → format string
  - %d → format specifier for int
  - a → value to be printed

scanf() function:

- scanf() is used to **take input** from the user through the keyboard.
- It is also defined in **stdio.h**.
- General syntax:

  scanf("format specifier", &variable_name);

- Example:

  int age;

  scanf("%d", &age);

- Here:
  - %d → we expect an integer
  - &age → address of variable age

Important difference:

- printf() → output function (displays data).
- scanf() → input function (reads data from user).

II. Give the difference between WHILE and DO…WHILE loop.

Point	while loop	do…while loop
1	**Entry-controlled loop**	**Exit-controlled loop**
2	Condition is checked **before** executing the body.	Body is executed **first**, then condition is checked.
3	If condition is false initially, loop body may **not execute even once**.	Body executes **at least once**, even if condition is false initially.
4	Syntax:	Syntax:

	c
	while(condition) {
	 // statements
	}
	
		c
		do {
		 // statements
		} while(condition);

Example while:

```
int i = 1;

while(i <= 5) {

   printf("%d ", i);

   i++;

}
```

Example do…while:

```
int i = 1;

do {

   printf("%d ", i);

   i++;

} while(i <= 5);
```

III. Explain any two string functions.

(Assume <string.h> is included.)

1. strlen()

- Used to **find the length of a string** (number of characters excluding '\0').
- Syntax:

  ```
  int strlen(char str[]);
  ```

- Example:

  char name[20] = "Atul";

  int len = strlen(name); // len = 4

2. strcpy()

- Used to **copy one string into another**.
- Syntax:

  char* strcpy(char dest[], const char src[]);

- Example:

  char s1[20] = "Hello";

  char s2[20];

  strcpy(s2, s1); // s2 becomes "Hello"

IV. Define Syntax Error.

- A **syntax error** is an error that occurs when we do **not follow the rules (grammar) of the programming language**.
- The compiler **detects** syntax errors during **compilation time**.
- Because of syntax errors, the program **cannot be successfully compiled** until they are removed.
- Examples:
  - Missing semicolon:
  - printf("Hello") // error: ; missing
  -
  - Using a keyword as variable name
  - int for; // error
  -

V. What is Algorithm?

- An **algorithm** is a **step-by-step, finite and logical sequence of instructions** to solve a specific problem.
- It is written in **simple English-like language**, not in any particular programming language.
- Characteristics:
  - It must have a **starting** and **ending** point.
  - It must be **finite** (should complete in limited steps).

- It should be **unambiguous** (clear and not confusing).
- It should give the **correct output** for valid input.

Example (Algorithm to add two numbers):

1. Start
2. Read two numbers A and B
3. Compute SUM = A + B
4. Display SUM
5. Stop

VI. Write different ways of writing comments on program.

In C, comments are used to **improve readability** and are **ignored by the compiler**.

There are mainly **two types of comments**:

1. **Single-line comment**
   - Starts with //
   - Everything after // on that line is treated as a comment.
   - Example:
   - // This is a single-line comment
   - int a = 10; // variable a stores 10
   -
1. **Multi-line (block) comment**
   - Starts with /* and ends with */
   - Can span multiple lines.
   - Example:
   - /* This is a
   - multi-line
   - comment */ int a = 10;
   -

VII. What is the use of EOF?

- **EOF** stands for **End Of File**.
- It is a special **marker/constant** used in C to indicate that there is **no more data** to read from a file or input stream.
- It is generally defined as **-1** in stdio.h.
- Used mainly with functions like getchar(), fgetc(), fscanf(), etc., to detect the end of input.

Example:

```
int ch;

while ((ch = getchar()) != EOF) {

   putchar(ch);

}
```

Here:

- The loop continues reading characters until **EOF** is encountered (for example, when user signals end of input or when file ends).

Q.1 (b) List and describe the basic data types available in C.

In C programming, **data types define the type of data a variable can store**.
The major **basic (fundamental / primitive) data types** in C are:

1. Integer (int)

- Used to store **whole numbers** (positive or negative).
- Does not contain decimal values.
- Typically occupies **2 or 4 bytes** in memory depending on compiler.
- Example:

```
int age = 18;
```

2. Floating Point (float)

- Used to store **real numbers / decimal values**.
- Provides **single-precision** accuracy.
- Usually occupies **4 bytes**.
- Example:

```
float marks = 72.50;
```

3. Double (double)

- Used to store **large or high-precision decimal values**.
- It is **double-precision**, meaning it is more accurate than float.
- Generally occupies **8 bytes**.
- Example:

```
double pi = 3.14159265;
```

4. Character (char)

- Used to store **single characters** like ‘A’, ‘9’, ‘$’, etc.
- Occupies **1 byte**.
- Character values are stored using **ASCII codes** internally.
- Example:

```
char grade = 'A';
```

Q.2 (a) What is a compiler, and how does it differ from an interpreter? (7 Marks)

Compiler

- A **compiler** is a system software used to **translate the entire source code** (written in high-level language like C, C++, Java, etc.) into **machine code** (binary) at once.
- It performs:
  - translation,
  - error checking,
  - optimization,
  - and generating an executable file.

Key Characteristics:

1. Translates program **as a whole**.
2. Errors are shown **after complete scanning**.
3. Faster execution because executable file is created once.
4. Used in languages like **C, C++, Java**.

Example compiler: GCC (GNU C Compiler)

Interpreter

- An **interpreter** is a software that **translates and executes code line-by-line**.
- It reads one instruction, converts it into machine code, executes it, and then moves to the next line.

Key Characteristics:

1. Translates program **line-by-line**.
2. Displays errors **immediately when found**.

3. Slower execution because translation occurs every time code runs.
4. Used in languages like **Python, JavaScript, BASIC**.

Difference between Compiler and Interpreter

Basis	Compiler	Interpreter
Working	Translates **entire program at once**	Translates **one line at a time**
Error Handling	Shows error **after full scanning**	Shows error **immediately line-wise**
Speed	Execution is **fast**	Execution is **slow**
Output	Produces **executable file**	Does not produce an executable file
Memory Usage	More memory required	Less memory required
Example Languages	C, C++, Java	Python, JavaScript, BASIC

Q.2 (b) Define flowchart. Explain the importance of flowchart. Also describe various symbols of flowchart.

Definition of Flowchart

A **flowchart** is a **graphical representation** of a process, algorithm, or program logic using symbols connected with arrows to show the flow of control.

It visually describes **how a program works step-by-step**.
It is commonly used in planning, analysis, designing and documenting programs.

Importance of Flowchart

Flowcharts are important because:

1. Easy Understanding

- It provides a **clear visual structure** of an algorithm, helping anyone understand the logic easily.

2. Problem Analysis

- Helps programmers **analyze the problem logically** before coding.

3. Better Communication

- Acts as a **communication tool** between designers, developers, teachers, and students.

4. Debugging

- Errors or missing logic can be **detected before writing code**, saving time.

5. Documentation

- Flowcharts are useful as **reference material** for future maintenance or updates.

6. Program Development

- They **simplify coding**, because the logic is already organized.

Flowchart Symbols and Their Meanings

Terminator / Start-End Symbol

Shape: **Oval / rounded rectangle**
Use: Represents the **beginning or ending** of a flowchart.
Example text: Start / Stop

Input / Output Symbol

Shape: **Parallelogram**
Use: Represents **data input or output**.
Example:

- "Read A, B"
- "Display Result"

Process Symbol

Shape: **Rectangle**
Use: Shows **processing / computation steps**.
Example:

- SUM = A + B
- i = i + 1

Decision / Condition Symbol

Shape: **Diamond**
Use: Represents **decision making** or **conditions**.
Example:

- A > B?
- Output has two paths: Yes / No

Flow Lines / Arrows

Use: Show **direction of flow** or sequence of steps.

Connector Symbol

Shape: **Small circle**
Use: Used to **connect parts of flowchart** when it continues to another page or place.

Predefined Process / Function

Shape: **Rectangle with double-sided vertical lines**
Use: Indicates a **predefined module or function**.

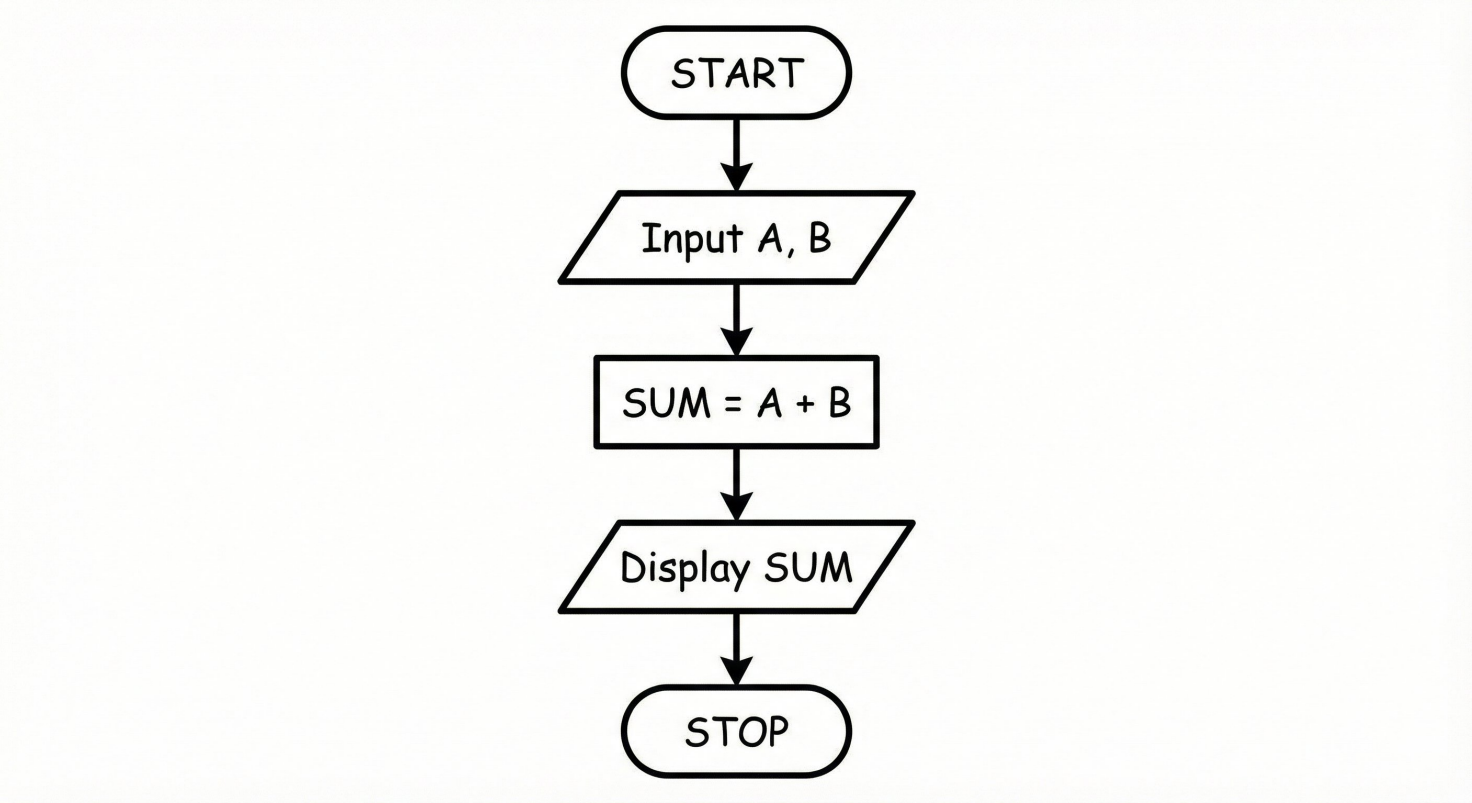
Off-Page Connector

Shape: **Pentagon**
Use: Indicates connection to **another flowchart** on a different page.

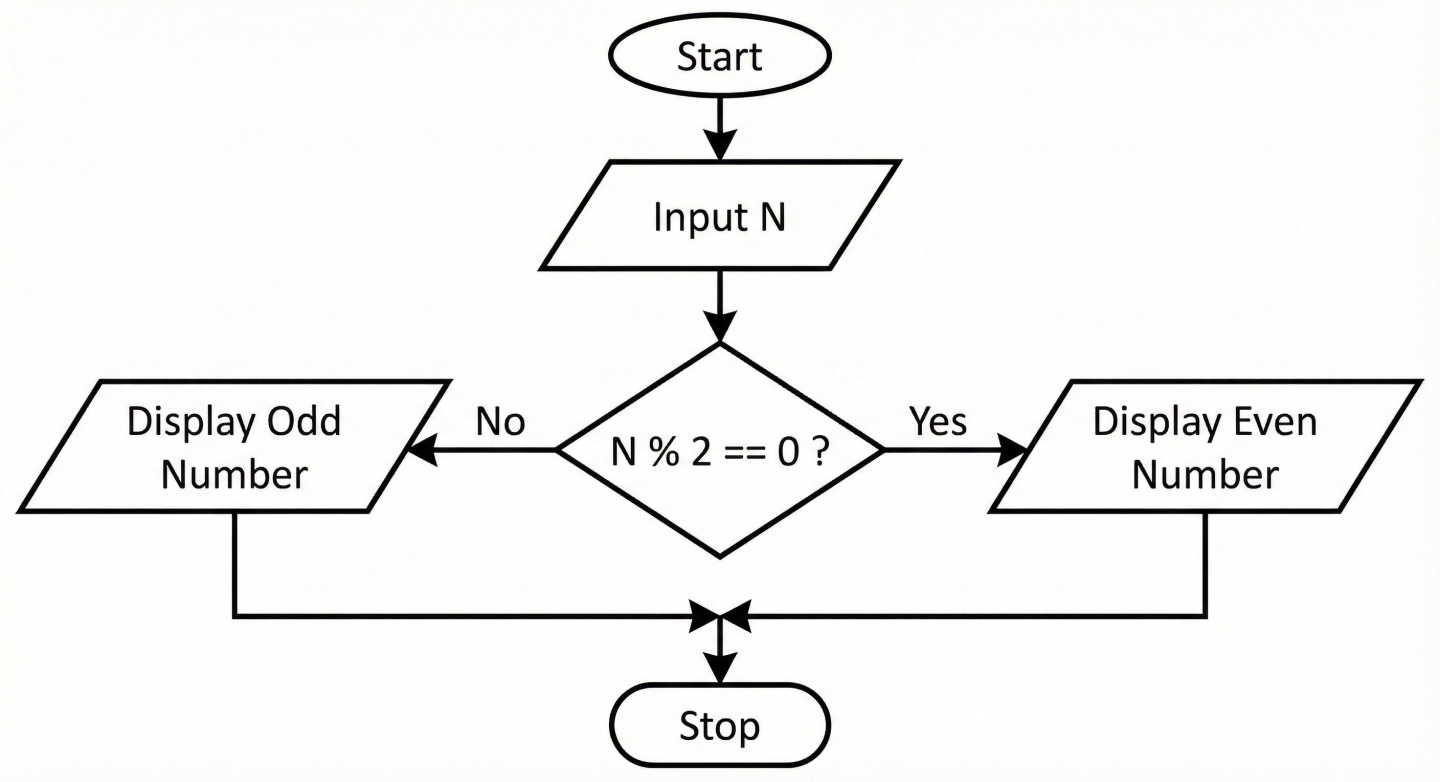
Simple Example (concept explanation):

For adding two numbers:

1. **Start**
2. **Input** A, B
3. **Process** SUM = A + B
4. **Output** SUM
5. **Stop**



A flowchart for checking odd or even number.



OR

Q.2(b) Upgraded Fully-Detailed Exam Answer

Structure of a C Program

A C program follows a **systematic and layered structure** to improve readability, execution order, modularity and debugging.

Below are the **essential components**:

1) Documentation Section

- Appears at the **beginning** of the program.
- Contains **comments** that describe:
  - program purpose
  - programmer name
  - creation/update date
  - logic explanation
- Used for **program documentation and clarity**.

Example:

/* Program to calculate average of three numbers */

2) Link Section

- Includes all **library/header files** needed.
- Using #include, we instruct the compiler to import predefined functions.
- Without headers, built-in functions like printf() may not work.

Example:

#include <stdio.h>

#include <math.h>

3) Definition Section

- Contains **symbolic/constant definitions** using #define.
- Improves code readability and reduces magic numbers.

Example:

#define PI 3.14

#define LIMIT 50

4) Global Declaration Section

- Variables or functions declared **outside main()** are considered *global*.
- Can be accessed by all functions in the program.
- Helpful for data sharing between modules.

Example:

int total;

5) main() Function Section

- This is the **entry point** of every C program.
- Execution always begins from main(), even if multiple functions exist.
- Consists of two internal parts:

a) Declaration Part

Used to declare local variables:

int a, b;

b) Executable Part

Contains statements that perform operations:

```
a = 10;

b = 20;

printf("%d", a + b);
```

6) Sub-program / Function Section

- Contains **user-defined functions**.
- Used for code modularity, reusability, easier testing, and readability.
- These functions can be invoked from main() or other functions.

Example:

```
void display()

{

   printf("Hello");

}
```

Program Structure Representation (Exam Friendly Layout)

Documentation Section → Comment lines describing program

Link Section → #include header files

Definition Section → #define constants

Global Declaration Section → Global variables / function prototypes

main() Function → Entry point + logic execution

User Defined Functions → Additional reusable functions

Q.3 (a) Explain the concept of type conversion and typecasting in C with example.

In C language, **different data types** have different sizes and ranges (like int, float, double, char, etc.).

Type Conversion (Implicit / Automatic Conversion)

Definition

Type conversion is the process in which **the compiler automatically converts** one data type into another **without the programmer explicitly asking**.

Isko **implicit type conversion** bhi kehte hain.

When does it happen?

Type conversion happens in situations like:

1. **Mixed expressions**
   (e.g. int + float, char + int etc.)
2. **Assignment** from one type to another
   (e.g. assigning float to int variable)

The compiler tries to **avoid data loss**, so it converts to a **higher / wider type**.

Type Conversion Rules in C (Simplified)

- If an operation involves:
  - int and float → int is converted to float
  - int and double → int is converted to double
  - char and int → char is converted to int
- In general, **lower type → higher type**
  char → int → float → double

Isko **"usual arithmetic conversions"** bhi bolte hain.

Example 1 – Integer + Float

```
int a = 10;

float b = 3.5;

float c;

c = a + b;
```

Explanation:

- a is int, b is float.
- Before addition, **a is automatically converted to float** → 10.0.
- Then: 10.0 + 3.5 = 13.5
- Result stored in float c.

Yaha programmer ne kahin (float)a nahi likha,
compiler ne khud hi type convert kiya → this is **type conversion**.

◈ Example 2 – Assignment Conversion

float x = 7.9;

int y;

y = x; // implicit conversion

Explanation:

- x is float, y is int.
- While assigning, **float → int** conversion hoti hai.
- 7.9 becomes 7. (decimal part lost)
- This can cause **loss of data**.

Typecasting (Explicit Type Conversion)

Definition

Typecasting is **manual conversion** of one data type into another using a **cast operator**.

Yaha programmer **khud decide karta hai** ki kis type me value ko treat karna hai.

Syntax:

(type) expression

Example:

(float) a

(int) b

(char) x

Example 1 – Avoid Integer Division

int a = 5, b = 2;

float result;

result = (float)a / b;

Without typecasting:

result = a / b; // 5 / 2 = 2 (int division) → result = 2.0

With typecasting:

result = (float)a / b; // 5.0 / 2 = 2.5 → result = 2.5

Explanation:

- (float)a converts a from int to float (5 → 5.0).
- Then 5.0 / 2 = 2.5 (floating point division).
- Yaha **typecasting programmer ne ki**, isliye result sahi aata hai.

Example 2 – Converting Float to Int

float x = 9.99;

int y;

y = (int)x; // explicit typecasting

Explanation:

- (int)x converts 9.99 → 9
- Decimal part is **truncated** (not rounded).
- Ye conversion programmer ke control me hai.

Example 3 – Char to Int (ASCII)

char ch = 'A';

int code;

code = (int)ch;

- Here code will store **65** (ASCII value of 'A').

Difference Between Type Conversion and Typecasting

Basis	Type Conversion (Implicit)	Typecasting (Explicit)
Who performs?	**Compiler automatically**	**Programmer manually**
Syntax	No special syntax	Uses (type) before expression
Control	Less control (compiler decides)	Full control (programmer decides)
Safety	Usually safer, tries to avoid data loss	Can cause data loss if not used carefully
Example	float c = a + b;	float c = (float)a / b;

Q.3 (b) Design a function in C that calculates the factorial of a number using recursion.

What is Factorial?

The factorial of a number **n** (written as n!) is:

n! = n × (n − 1) × (n − 2) × ... ×

Example:

5! = 5 × 4 × 3 × 2 × 1 = 120

Special case:

0! = 1

What is Recursion?

Recursion is a programming technique where:

a function calls itself to solve a smaller version of a problem.

Using recursion, factorial can be defined as:

fact(n) = 1 if n == 0

fact(n) = n × fact(n − 1) if n > 0

Recursive Function Logic

Base Case:
Stops recursion

if n == 0 → return 1

Recursive Case:

return n * factorial(n-1)

C Program to Calculate Factorial Using Recursion

```
#include <stdio.h>

// Recursive function declaration

int factorial(int n)

{

   if(n == 0)
```

```
        return 1;      // Base condition
    else
        return n * factorial(n - 1);  // Recursive call
}
int main()
{
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Factorial of %d is %d", num, factorial(num));
    return 0;
}
```

Step-by-Step Execution (For n = 5)

Function calls:

factorial(5)

→ 5 × factorial(4)

→ 4 × factorial(3)

→ 3 × factorial(2)

→ 2 × factorial(1)

→ 1 × factorial(0)

→ returns 1

Then result returns back:

1 → 1 × 1 = 1

1 × 2 = 2

2 × 3 = 6

6 × 4 = 24

24 × 5 = 120

Final output: **120**

Program Output Example

Enter a number: 5

Factorial of 5 is 120

Q.3 (a) Create a simple program using conditional execution (if-else statements) to determine if a number is odd or even

Concept Understanding

A number is **even** if it is divisible by 2
(i.e., remainder = 0)

n % 2 == 0 → Even

Otherwise, it is **odd**.

Else → Odd

Logic / Algorithm (Exam-friendly stepwise explanation)

Start

Input number n

Check condition:

👉 if n % 2 == 0

- print "Even Number"

Else

- print "Odd Number"

Stop

C Program Using If–Else

```
#include <stdio.h>

int main()

{

   int num;

   printf("Enter a number: ");

   scanf("%d", &num);

   if(num % 2 == 0){

      printf("%d is an Even Number", num);

   }else{

      printf("%d is an Odd Number", num);

}

   return 0;

}
```

Explanation of the Program

Variable num takes user input
Expression num % 2 finds remainder
If remainder = 0 → number **is even**
Else → number **is odd**
Decision made using **if-else statement**

Example Output

Enter a number: 7

7 is an Odd Number

Enter a number: 12

12 is an Even Number

Q.3 (b) Explain Entry and Exit Loops with Example.

Looping statements allow repetition of instructions multiple times.
C me loops ko **two major categories** me divide kiya jata hai:

Entry-controlled loops
Exit-controlled loops

1) Entry-Controlled Loop

Definition

A loop in which **condition is checked first**, before executing loop body is called **Entry-controlled loop**.

Agar condition **true hai tabhi** loop body execute hoti hai.
Agar condition **false ho jaye**, body execute nahi hoti.

Examples in C:

for loop
while loop

Example (while loop – Entry controlled)

```
int i = 1;

while(i <= 5)

{

   printf("%d ", i);

   i++;

}
```

Explanation

- Condition i <= 5 pehle check hoti hai.
- Agar condition false ho, toh body run **nahi hoti**.
- i.e., Entry-controlled loops **may execute zero times**.

2) Exit-Controlled Loop

Definition

A loop in which **body is executed first** and condition is checked **after execution** is called **Exit-controlled loop**.

Loop body **at least one time execute hoti hi hoti hai**, chahe condition false ho.

Example in C:

do…while loop

Example (do…while loop – Exit controlled)

```
int i = 1;

do

{

   printf("%d ", i);

   i++;

}

while(i <= 5);
```

Explanation

- Body executes **first**, then condition checked.
- Even if condition becomes false, **loop runs at least once**.

Difference Between Entry and Exit Loops

Basis	Entry Controlled Loops (for, while)	Exit Controlled Loop (do-while)
Condition check	before execution	after execution
Minimum execution	may execute 0 times	executes at least 1 time
Type	pre-test loop	post-test loop
Syntax example	while(condition)	do{ } while(condition);

Q.4 (a) Explain four different category of Function based on argument and return type.

In C programming, functions are classified into **four categories** based on:

1. **Whether they take arguments (parameters)**
2. **Whether they return a value**

Ye 4 categories hain:

1. **No arguments and no return value**

2. **Arguments but no return value**
3. **No arguments but return value**
4. **Arguments and return value**

1) Function with No Arguments and No Return Value

Definition

- Function **does not take any input (no parameters)**
- Function **does not return any value** to the calling function
- Mostly used when:
  - data is taken **inside the function** itself (e.g., using scanf)
  - and result is **directly printed** from inside the function

General Syntax

```
void function_name()

{

   // statements

}
```

Example

```
#include <stdio.h>

void add()   // no argument, no return type

{

   int a, b, sum;

   printf("Enter two numbers: ");

   scanf("%d %d", &a, &b);

   sum = a + b;

   printf("Sum = %d", sum);

}

int main()

{

   add();   // function call
```

```
    return 0;
}
```

Explanation

- add() doesn't accept anything from main()
- add() doesn't return anything to main()
- It **takes input and prints output** within itself
- Return type void → no value returned

2) Function with Arguments but No Return Value

Definition

- Function **takes arguments (parameters)** from the calling function
- But it **does not return any value** (return type is void)
- Used when:
    - data is prepared in main() and **passed** to function
    - function only **performs some task** (like printing, updating, etc.)

General Syntax

```
void function_name(type arg1, type arg2)
{
    // statements using arg1, arg2
}
```

Example

```
#include <stdio.h>
void add(int x, int y)   // arguments, no return value
{
    int sum;
    sum = x + y;
    printf("Sum = %d", sum);
}
```

```
int main()
{
   int a = 10, b = 20;
   add(a, b);   // values passed as arguments
   return 0;
}
```

Explanation

- a and b are passed to add() as **arguments**
- add() calculates and **prints** the sum
- No value is returned to main() → return type void

3) Function with No Arguments but Return Value

Definition

- Function **does not take any arguments**
- But it **returns a value** to the calling function
- Used when:
    - function internally **takes input/does calculation**
    - and returns a **final result** to the caller

General Syntax

```
type function_name()
{
   // statements
   return value;
}
```

Example

```
#include <stdio.h>
int getNumber()   // no argument, returns int
{
   int n;
```

```
    printf("Enter a number: ");
    scanf("%d", &n);
    return n;
}
int main()
{
    int x;
    x = getNumber();    // function returns value
    printf("You entered: %d", x);
    return 0;
}
```

Explanation

- getNumber() does **not receive arguments**
- Inside function, it **reads value from user**
- Then return n; sends value back to main()
- main() stores it in variable x

4) Function with Arguments and Return Value

Definition

- Function **takes arguments**
- Function **returns a value**
- This is the **most useful and flexible type**
- Used for:
    - performing operations on data given by caller
    - and sending back **result**

General Syntax

```
type function_name(type arg1, type arg2)
{
    // statements
```

```
    return value;
}
```

Example

```
#include <stdio.h>
int add(int x, int y)   // arguments and return value
{
    int sum;
    sum = x + y;
    return sum;   // return result
}
int main()
{
    int a = 10, b = 20, result;
    result = add(a, b);   // function call with arguments
    printf("Sum = %d", result);
    return 0;
}
```

Explanation

- a and b are passed to add()
- Function add() calculates sum and **returns** it
- Returned value is stored in result in main()
- main() prints the sum

Q.4 (b) Explain Union with proper example. What is the difference between Union & Structure.

What is a Union?

A **Union** in C is a **user-defined data type** similar to structure,
BUT — in a union, **all members share the same memory location**.

Means, **only one member stores value at a time**
Union ka size = **largest member size**

Declared using keyword union

Syntax of Union

```
union union_name
{
   data_type member1;
   data_type member2;
   ...
};
```

Example:

```
union student
{
   int rollno;
   float marks;
   char grade;
};
```

Why Union is Used?

✔ To **save memory**
✔ Useful when different values are stored at different times
✔ Used in system-level programming (e.g., device drivers, communication packets)

Example Program of Union

```
#include <stdio.h>

union student

{

   int rollno;

   float marks;

};

int main()

{

   union student s1;

   s1.rollno = 101;     // stores roll number

   printf("Roll Number = %d\n", s1.rollno);

   s1.marks = 89.50;     // now stores marks

   printf("Marks = %.2f\n", s1.marks);

   // NOTE: rollno value is lost because same memory is reused

   return 0;

}
```

Explanation of Output

- First rollno = 101 is stored
- Then marks = 89.50 is stored
- Kyunki dono members **same memory share karte hain**,
  ✔ marks assign hone ke baad rollno value **overwritten** ho jati hai

Therefore unions allow storing **only one valid value at a time**

Memory Size Concept of Union

Example union:

```
union test {
```

```
    int a;      // 4 bytes
    double b;   // 8 bytes
    char c;     // 1 byte
};
```

Size = **largest member** → double = 8 bytes

NOT sum of all members

Difference Between Structure and Union

Feature	Structure	Union
Memory allocation	Each member gets **separate memory**	**All members share same memory**
Storage	**Multiple members** can store values at same time	Only **one member** holds valid value at a time
Size (example)	Sum of all members	Size of **largest member only**
Usage	Used when we need to store different values together	Used when we need to store different values **but one at a time**
Performance	Slightly slower (more memory)	Faster and memory-efficient
Updating values	Doesn't affect other members	Updating one member **overwrites** other values

OR

Q.4 (a) What is a string? List out all built-in string functions and Write a C Program for comparing two strings.

What is a String

A **string** in C is a sequence of characters **ending with null character '\0'**.
It is stored as a **character array**.

Example declaration:

```
char name[20] = "Atul";
```

Memory representation:

A t u l \0

✔ Strings are stored using char[]

✔ Must contain **terminating null character** (\0)

Built-in String Functions in <string.h>

Here are important C string functions:

1. strlen(s) → returns length of a string
2. strcpy(dest, src) → copies string
3. strcat(s1, s2) → concatenates two strings
4. strcmp(s1, s2) → compares two strings
5. strrev(s) → reverses a string (compiler dependent)
6. strupr(s) → converts to uppercase (compiler dependent)
7. strlwr(s) → converts to lowercase (compiler dependent)

Standard ones: strlen, strcpy, strcat, strcmp

⋆ Program: Compare Two Strings

```
#include <stdio.h>

#include <string.h>

int main()

{

   char s1[20], s2[20];

   printf("Enter first string: ");

   gets(s1);

   printf("Enter second string: ");

   gets(s2);

   if(strcmp(s1, s2) == 0)

      printf("Strings are equal.");

   else
```

```
        printf("Strings are not equal.");
    return 0;
}
```

Explanation:

- gets() reads string input
- strcmp(s1, s2) → returns:
  - **0** if strings equal
  - **+ve / -ve** if different
- If equal → prints **"Strings are equal"**

Q.4 (b) Explain 2-D array with examples.

Definition

A **2-D array** is a collection of elements arranged in **rows and columns** (like a table or matrix).

Example:

int a[3][4];

Meaning → **3 rows, 4 columns**

Memory is stored **row-wise (row-major order)**.

Declaration Methods

int arr[2][3];

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

Initialization Example

```
int mat[2][2] = {
    {10, 20},
    {30, 40}
};
```

Stored as:

10	**20**
30	40

Accessing Elements

Syntax:

array[row][column]

Example:

int x = mat[1][0]; // 2nd row, 1st column (value 30)

Program Example — Input & Display a Matrix

```
#include <stdio.h>

int main()

{

   int a[2][2], i, j;

   printf("Enter 4 elements:\n");

   for(i = 0; i < 2; i++)

   {

      for(j = 0; j < 2; j++)

      {

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

      }

   }

   printf("Matrix is:\n");

   for(i = 0; i < 2; i++)

   {

      for(j = 0; j < 2; j++)

      {

         printf("%d ", a[i][j]);

      }
```

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

Explanation

- First nested loop → stores values
- Second nested loop → prints values row-wise like table

Q.5 (a) What is a Pointer? Explain array of pointer with suitable example.

What is a Pointer

A **pointer** is a special variable that **stores the address of another variable**.

Instead of storing a value directly, it stores the **memory location** where the value exists.
It allows **indirect access** to data.

Syntax:

data_type *pointer_name;

Example:

int *p;

Here, p stores the **address of an integer variable**.

Why are Pointers Used?

✔ Efficient memory usage
✔ Dynamic memory allocation
✔ Fast access to arrays and strings
✔ Function argument passing by reference

Example of Pointer Usage

int a = 10;

int *p;

p = &a;

```
printf("Value: %d", *p);     // Indirect value access

printf("Address: %p", p);    // Address stored in pointer
```

✔ &a gives address of variable a

✔ *p accesses value stored at that address

Array of Pointers

Definition

An **array of pointers** is an array where **each element is a pointer** instead of normal data.

✔ Used when you want to store addresses of multiple variables

✔ Useful in string handling, dynamic arrays, function references

Example: Array of Integer Pointers

```
#include <stdio.h>

int main()

{

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

   int *ptr[3];    // array of 3 pointers

   ptr[0] = &a;

   ptr[1] = &b;

   ptr[2] = &c

   printf("Values:\n");

   printf("%d\n", *ptr[0]);

   printf("%d\n", *ptr[1]);

   printf("%d\n", *ptr[2]);

   return 0;

}
```

Explanation:

- ptr is an array of 3 integer pointers
- ptr[0] stores address of a
- ptr[1] stores address of b
- ptr[2] stores address of c

Using *ptr[i], we **indirectly access values** 10, 20, 30

Q.5 (b) Describe how to define and use a structure in C.

What is Structure

A **structure** in C is a user-defined data type that groups **different types of variables** under one name.

Example: A student has
name (string), rollno (int), marks (float).
Structure allows storing all these in **one unit**.

Syntax for Defining a Structure

```
struct structure_name
{
   data_type member1;
   data_type member2;
   ...
};
```

Example:

```
struct student
{
   int rollno;
   float marks;
   char name[20];
};
```

Declaring Structure Variables

Two ways:

1. After structure definition:

```
struct student s1, s2;
```

2. Inside structure declaration:

```
struct student
{
   int rollno;
   float marks;
} s1, s2;
```

Initializing Structure Members

```
struct student s1 = {101, 89.5, "Atul"};
```

Values are assigned in **declared order**

Accessing Structure Members

We use **dot operator** (.)

```
printf("%d", s1.rollno);
printf("%f", s1.marks);
printf("%s", s1.name);
```

Full Example Program — Using Structure

```
#include <stdio.h>
struct student
{
   int rollno;
   float marks;
   char name[20];
};
```

```
int main()
{
    struct student s1;
    printf("Enter roll number: ");
    scanf("%d", &s1.rollno);
    printf("Enter marks: ");
    scanf("%f", &s1.marks);
    printf("Enter name: ");
    scanf("%s", s1.name);
    printf("\nStudent Details:\n");
    printf("Roll No: %d\n", s1.rollno);
    printf("Marks: %.2f\n", s1.marks);
    printf("Name: %s\n", s1.name);
    return 0;
}
```

Explanation

✔ Structure defines **multiple grouped attributes**
✔ s1 is a structure variable
✔ Dot operator accesses internal members
✔ Data is stored/displayed in an organized format

OR

Q.5 (a) Explain the process of declaring, opening, and closing a file in C with example.

Introduction to File Handling

In C, file handling allows us to **store and access data permanently** on disk through operations like:

✔ create
✔ read

✔ write

✔ append

Files are handled using **FILE pointer** given by <stdio.h>.

1) Declaring a File Pointer

Before working with a file, we declare a pointer of type FILE.

Syntax:

FILE *fp;

fp will store reference of file being used.

2) Opening a File

We open a file using fopen():

fp = fopen("filename.ext", "mode");

File Modes:

Mode	Purpose
"r"	Read existing file
"w"	Write (create new or overwrite)
"a"	Append (add data to end)
"r+"	Read + write
"w+"	Read + write (overwrite)
"a+"	Read + write (append)

3) Closing a File

Once work is finished, we close file using:

fclose(fp);

✔ This frees memory and ensures data is properly saved.

Example Program — File Open, Write & Close

```
#include <stdio.h>

int main()

{
```

```
    FILE *fp;
    fp = fopen("demo.txt", "w");   // open file for writing
    if(fp == NULL)
    {
       printf("File not found!");
       return 0;
    }
    fprintf(fp, "Hello GTU!");     // write data to file
    fclose(fp);                    // close file
    printf("Data written successfully.");
    return 0;
}
```

Explanation:

- Declare file pointer → FILE *fp
- Open file → fopen("demo.txt", "w")
- Write data → fprintf()
- Close file → fclose(fp)

Q.5 (b) What are the differences between working with text files and binary files in C?

Definition of Text File

A text file stores data in **human-readable form**,
characters follow ASCII representation.

Example: .txt, .csv

Definition of Binary File

A binary file stores data in **machine-readable encoded form**
and not easily readable by humans.

Example: .exe, .dat, .bin

Differences Between Text File and Binary File

Feature	Text File	Binary File
Format	Human-readable text (characters)	Machine-readable encoded form
Size	Larger size (ASCII overhead)	Smaller size (compact storage)
Storage Method	Stores ASCII characters	Stores raw binary bits
Access	Read using standard text editors	Can’t be viewed normally
Accuracy	Possible loss due to formatting	More accurate representation
Functions Used	fprintf(), fscanf(), fgets()	fread(), fwrite()
Speed	Slower processing	Faster data access
Best for	Simple text data	Large numeric/structured data

Example Writing to Text File

```
fprintf(fp, "Roll No = %d", roll);
```

Data stored as:

R o l l ... 1 0 1

Example Writing to Binary File

```
fwrite(&roll, sizeof(int), 1, fp);
```

Data stored as compact raw bits (not readable).