1. What is a function prototype? Why is it required in programming? Discuss with an example program that demonstrates the use of a function prototype. (9)

Definition:

A **function prototype** is a declaration of a function that tells the compiler about the function name, return type, and parameters before its actual definition. It helps in ensuring type checking and prevents errors due to implicit declarations.

Why is a Function Prototype Required?

Ensures Type Checking: The compiler checks the function calls for correct argument types and order.

Allows Function Calls Before Definition: It enables calling a function before its actual definition.

Improves Readability & Maintainability: Provides a clear structure of the program.

Prevents Implicit Declaration Errors: Without a prototype, the compiler assumes the function returns an int, leading to possible incorrect behavior.

```c
#include <stdio.h>

// Function prototype
int add(int, int);

int main() {
    int num1 = 5, num2 = 10, sum;

    // Function call before definition
    sum = add(num1, num2);

    printf("Sum: %d\n", sum);

    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

2. Write a program to demonstrate the concepts of function prototype, function call, and function definition. Explain the flow of execution in the program (9)

```c
#include <stdio.h>

// Function Prototype (Declaration)
int multiply(int, int);

int main() {
    int num1 = 5, num2 = 10, result;

    // Function Call
    result = multiply(num1, num2);

    // Displaying the result
    printf("The product of %d and %d is: %d\n", num1, num2, result);

    return 0;
}

// Function Definition
int multiply(int a, int b) {
    return a * b;
}
```

Explanation of the Flow of Execution

1. **Function Prototype (Declaration)**
   - `int multiply(int, int);`
   - This tells the compiler that a function named `multiply` exists, returns an `int`, and takes two `int` arguments.
   - This ensures the function can be called before its actual definition.
2. **Main Function Execution**
   - The `main()` function starts execution.
   - Two integers, `num1` and `num2`, are initialized with values 5 and 10.
   - The function `multiply(num1, num2);` is called.

3. **Function Call**
   - When `multiply(num1, num2)` is encountered, control jumps to the function definition.
   - The values `5` and `10` are passed as arguments (`a = 5`, `b = 10`).
4. **Function Definition Execution**
   - The function `multiply()` executes: `return a * b;`
   - It calculates `5 * 10 = 50` and returns `50` to the calling function.
5. **Back to Main Function**
   - The returned value `50` is stored in `result`.
   - `printf()` prints the output:

3. Explain in detail what a function definition is, along with its syntax and key components. Provide an example of a function that calculates the factorial of a number, and explain each part of the function. (9)

Function Definition in C

A **function definition** is the actual implementation of a function. It contains the **function header** (return type, name, parameters) and the **function body** (statements that define what the function does). The function executes only when it is called.

```
return_type function_name(parameter1_type parameter1, parameter2_type parameter2, ...) {
    // Function body (statements)
    return value; // (if return_type is not void)
}
```

Key Components of a Function Definition

1. **Return Type:** Specifies the type of value the function returns.
   - Example: `int`, `float`, `char`, `void` (if no value is returned).
2. **Function Name:** The identifier used to call the function.
3. **Parameters (Optional):** Inputs to the function enclosed in parentheses.
   - If no parameters, use `void` or empty parentheses `()`.
4. **Function Body:** Contains the executable statements.
5. **Return Statement (Optional):** Returns a value to the calling function if needed.

Example: Factorial Function

```c
#include <stdio.h>

// Function prototype (declaration)
long long factorial(int n);

int main() {
    int num;

    // User input
    printf("Enter a number: ");
    scanf("%d", &num);

    // Function call
    if (num < 0)
        printf("Factorial is not defined for negative numbers.\n");
    else
        printf("Factorial of %d is %lld\n", num, factorial(num));

    return 0;
}

// Function definition
long long factorial(int n) {
    long long fact = 1; // To store the result

    for (int i = 1; i <= n; i++) {
        fact *= i; // Multiply fact by i
    }

    return fact; // Return the factorial value
}
```

Explanation of Each Part

1. **Function Prototype** (`long long factorial(int n);`)
   - Declares the function before `main()` to ensure proper type checking.
2. **Function Call** (`factorial(num)`)
   - Calls the function with user input.
3. **Function Definition**
   - **Return Type:** `long long` (factorial values can be large).
   - **Function Name:** `factorial`
   - **Parameter:** `int n` (input number).
   - **Body:**
     - Initializes `fact = 1` to store the result.
     - Uses a `for` loop to multiply numbers from `1` to `n`.
     - Returns the computed factorial.

4. Write a function prototype for a function that takes two integers and returns a float. Explain with an example.

float divide(int, int);

- **Return Type:** `float` (since division may result in a decimal value).
- **Parameters:** Two `int` values (numerator and denominator).

```c
#include <stdio.h>

// Function prototype
float divide(int, int);

int main() {
    int a = 10, b = 3;
    printf("Result: %.2f\n", divide(a, b)); // Function call
    return 0;
}

// Function definition
float divide(int x, int y) {
    return (float)x / y; // Type casting for float division
}
```

5. What is a function prototype? Why is it important?

A **function prototype** is a declaration that informs the compiler about the function's name, return type, and parameters **before** its actual definition.

Importance of Function Prototype:

1. **Ensures Type Checking:** Prevents incorrect function calls.
2. **Allows Function Calls Before Definition:** Enables better program organization.
3. **Improves Code Readability and Maintainability.**
4. **Prevents Implicit Declaration Errors:** Avoids issues due to missing return types or parameters.

6. Explain the difference between a function definition and a function declaration.

FEATURE	FUNCTION DECLARATION (PROTOTYPE)	FUNCTION DEFINITION
Purpose	Declares the function before use.	Implements the function logic.
Includes Function Body?	❌ No	✅ Yes
Return Type & Parameters?	✅ Specified	✅ Specified
Execution	❌ Does not execute	✅ Executes when called
Example	`int multiply(int, int);`	`int multiply(int a, int b) { return a * b; }`

7.

a) Explain parameter passing by reference in C with an example. (5 marks)

In **pass-by-reference**, a function receives the **address (memory location)** of the actual arguments instead of copies. This allows the function to modify the original values.

Eg)

```c
#include <stdio.h>

// Global variables
int a = 10, b = 20;

// Function to swap values (modifies global variables)
void swap() {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    printf("Before swapping: a = %d, b = %d\n", a, b);

    swap(); // Function call (modifies global variables)

    printf("After swapping: a = %d, b = %d\n", a, b);
    return 0;
}
```

b) Write a program to demonstrate parameter passing by reference by swapping two variables. (4 marks)

```c
#include <stdio.h>

// Global variables
int a = 10, b = 20;

// Function to swap values (modifies global variables)
void swap() {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    printf("Before swapping: a = %d, b = %d\n", a, b);

    swap(); // Function call (modifies global variables)

    printf("After swapping: a = %d, b = %d\n", a, b);
    return 0;
}
```

8.a) Explain parameter passing by value in C with an example. (5 marks)

Pass-by-value means that a function receives **a copy** of the actual parameter's value. Any changes made to the parameter inside the function **do not affect** the original value in the calling function.

```c
#include <stdio.h>

// Function that modifies the parameter (but does not affect the original)
void modify(int num) {
    num = num + 10;  // This change will not affect the original value
    printf("Inside function: num = %d\n", num);
}

int main() {
    int value = 5;
    printf("Before function call: value = %d\n", value);

    modify(value); // Function call (passes a copy)

    printf("After function call: value = %d\n", value); // Original value remains unchanged
    return 0;
}
```

b) Write a program to demonstrate parameter passing by value by swapping two variables. (4 marks)

```c
#include <stdio.h>

// Function to swap two numbers (does not work as expected)
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;

    printf("Before swapping: a = %d, b = %d\n", a, b);

    swap(a, b); // Function call (values are passed by value)

    printf("After swapping: a = %d, b = %d\n", a, b); // No change
    return 0;
}
```

9.Discuss the difference between call by value and call by reference parameter passing techniques with the help of suitable examples.(9)

Example: Call by Value

```c
#include <stdio.h>

// Function using call by value
void modify(int num) {
    num = num + 10;  // This change will not reflect in the original variable
    printf("Inside function: num = %d\n", num);
}

int main() {
    int value = 5;
    printf("Before function call: value = %d\n", value);

    modify(value); // Function call (passes a copy)

    printf("After function call: value = %d\n", value); // No change in original variable
    return 0;
}
```

2. Call by Reference

```c
#include <stdio.h>

// Global variables
int a = 10, b = 20;

// Function modifies global variables
void swap() {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    printf("Before swapping: a = %d, b = %d\n", a, b);

    swap(); // Function modifies global variables

    printf("After swapping: a = %d, b = %d\n", a, b);
    return 0;
}
```

FEATURE	CALL BY VALUE	CALL BY REFERENCE (WITHOUT POINTERS)
Parameter Type	Copy of variable	Global variables or structures
Effect on Original Variable	**No change**	**Changes reflected**
Memory Usage	More (extra copy created)	Less (no duplicate variables)
Security	Safe (original data unchanged)	**Risky** (global variables affect entire program)
Best Used For	When we don't want to modify original values	When we need to modify original values
Example	`void func(int a)`	`void swap() { global_var = new_value; }`

10. Write a C program with a function that finds the largest of two numbers.

```c
#include <stdio.h>

// Function to find the largest of two numbers
int findLargest(int a, int b) {
    return (a > b) ? a : b; // Returns the larger number
}

int main() {
    int num1, num2;

    // Taking input from user
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    // Function call
    int largest = findLargest(num1, num2);

    // Output the result
    printf("The largest number is: %d\n", largest);
    return 0;
}
```

11. Define formal parameters and actual parameters. Illustrate with an example.

Formal Parameters:

- These are the parameters defined in the function declaration or definition.
- They act as **placeholders** for values passed to the function.

Actual Parameters:

- These are the **real values** passed to the function when it is called.
- They are substituted into the **formal parameters** at runtime.

```c
#include <stdio.h>

// Function with formal parameters (x, y)
void displaySum(int x, int y) {
    printf("Sum = %d\n", x + y);
}

int main() {
    int a = 5, b = 10;

    // Function call with actual parameters (a, b)
    displaySum(a, b);

    return 0;
}
```

12. Name the different types of parameter passing. Illustrate each of them with an example.

1. **Pass by Value**
    - A copy of the variable is passed.
    - Changes inside the function **do not** affect the original variable.

```c
#include <stdio.h>

void modify(int num) {
    num = num + 10; // Changes only the local copy
}

int main() {
    int value = 5;
    modify(value);
    printf("After function call: %d\n", value); // Remains 5
    return 0;
}
```

Pass by Reference (Using Global Variables - Without Pointers)

- The function modifies a **global variable**, affecting the original value.

```c
#include <stdio.h>

int globalVar = 5; // Global variable

void modify() {
    globalVar = globalVar + 10; // Modifies original variable
}

int main() {
    modify();
    printf("After function call: %d\n", globalVar); // Changes to 15
    return 0;
}
```

3. **Pass by Reference (Using Structures - Without Pointers)**
    - Instead of modifying values in place, we return a **structure** with the new values.

```c
#include <stdio.h>

struct Result {
    int newValue;
};

struct Result modify(int num) {
    struct Result res;
    res.newValue = num + 10;
    return res;
}

int main() {
    int value = 5;
    struct Result result = modify(value);
    printf("After function call: %d\n", result.newValue); // Changes to 15
    return 0;
}
```

13. Define recursion with an example. Differentiate between iteration and recursion.

Definition:

Recursion is a process where a function **calls itself** directly or indirectly to solve a problem. It is mainly used when a problem can be **broken down into smaller subproblems** of the same type.

```c
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

Difference Between Iteration and Recursion

FEATURE	RECURSION	ITERATION
Definition	Function calls itself	Loop repeats a block of code
Memory Usage	Uses stack memory (function calls)	Uses loop control variables (less memory)
Speed	Slower (due to overhead of function calls)	Faster (no extra function calls)
Complexity	Used for problems that are naturally recursive (e.g., trees, Fibonacci)	Used for simple loops and repetitive tasks
Termination	Stops when base case is reached	Stops when loop condition becomes false
Example (Factorial)	`return n * factorial(n-1);`	`for(i=1; i<=n; i++) result *= i;`

14. What is recursion? Write a C program to display the Fibonacci series using a recursive function.

Definition:

Recursion is a process where a function **calls itself** directly or indirectly to solve a problem. It is mainly used when a problem can be **broken down into smaller subproblems** of the same type.

```c
#include <stdio.h>

// Recursive function to find Fibonacci numbers
int fibonacci(int n) {
    if (n == 0) return 0;  // Base case 1
    if (n == 1) return 1;  // Base case 2
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}

int main() {
    int terms;
    printf("Enter the number of terms: ");
    scanf("%d", &terms);

    printf("Fibonacci Series: ");
    for (int i = 0; i < terms; i++) {
        printf("%d ", fibonacci(i)); // Calling recursive function
    }

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

15. What is recursion? Write a C program to find the factorial of a given number using a recursion method.

Repeat

```c
#include <stdio.h>

// Recursive function to calculate factorial
long long factorial(int n) {
    if (n == 0) // Base case
        return 1;
    return n * factorial(n - 1); // Recursive call
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    // Function call and output
    printf("Factorial of %d is %lld\n", num, factorial(num));

    return 0;
}
```

16. Write a C program to find the sum of the first n natural numbers using recursion.

```c
#include <stdio.h>

// Recursive function to calculate sum of first n natural numbers
int sumNatural(int n) {
    if (n == 0) // Base case
        return 0;
    return n + sumNatural(n - 1); // Recursive call
}

int main() {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);

    printf("Sum of first %d natural numbers is: %d\n", n, sumNatural(n));

    return 0;
}
```

17. What is recursion? Give an example.

Definition:

Recursion is a process where a function **calls itself** directly or indirectly to solve a problem. It is mainly used when a problem can be **broken down into smaller subproblems** of the same type.

```c
#include <stdio.h>

// Recursive function to find factorial
int factorial(int n) {
    if (n == 0) return 1; // Base case
    return n * factorial(n - 1); // Recursive call
}

int main() {
    printf("Factorial of 5 is: %d\n", factorial(5));
    return 0;
}
```

18. Write a C program to find the sum of digits of a number using recursion.

```c
#include <stdio.h>

// Recursive function to calculate sum of digits
int sumOfDigits(int n) {
    if (n == 0) return 0; // Base case
    return (n % 10) + sumOfDigits(n / 10); // Recursive call
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    printf("Sum of digits of %d is: %d\n", num, sumOfDigits(num));

    return 0;
}
```

19. Write a program to demonstrate passing a one-dimensional array to a function. The function should return the largest element in the array.

```c
#include <stdio.h>

// Function to find the largest element in an array
int findLargest(int arr[], int size) {
    int max = arr[0]; // Assume first element is largest

    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i]; // Update max if a larger element is found
        }
    }
    return max;
}

int main() {
    int n;

    // Taking array size input
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Taking array elements input
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Function call to find the largest element
    int largest = findLargest(arr, n);

    // Output result
    printf("The largest element in the array is: %d\n", largest);

    return 0;
}
```

20. Write a C program to sort N numbers using functions

```c
#include <stdio.h>
// Function to sort an array using Bubble Sort
void sortArray(int arr[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) { // Swap if the element is greater
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
// Function to display the array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n;
    // Input: Number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    // Input: Array elements
    printf("Enter %d numbers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    // Sorting function call
    sortArray(arr, n);

    // Output: Sorted array
    printf("Sorted array: ");
    displayArray(arr, n);
    return 0;
}
```

21. Write a C program to find the second largest element of an array using user defined functions

```
#include <stdio.h>

// Function to find the second largest element
int findSecondLargest(int arr[], int size) {
    int largest, secondLargest;

    if (size < 2) {
        printf("Array should have at least two elements.\n");
        return -1;
    }

    largest = secondLargest = -2147483648; // Smallest possible integer value

    for (int i = 0; i < size; i++) {
        if (arr[i] > largest) {
            secondLargest = largest; // Update second largest
            largest = arr[i]; // Update largest
        } else if (arr[i] > secondLargest && arr[i] != largest) {
            secondLargest = arr[i]; // Update second largest
        }
    }

    if (secondLargest == -2147483648) {
        printf("No second largest element found.\n");
        return -1;
    }

    return secondLargest;
}

int main() {
    int n;

    // Input: Number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Input: Array elements
    printf("Enter %d numbers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Function call to find the second largest element
    int secondLargest = findSecondLargest(arr, n);
```

```
    // Output result
    if (secondLargest != -1) {
        printf("The second largest element is: %d\n", secondLargest);
    }

    return 0;
}
```

22. Write a program to find the average of n elements of an array using a function.

```c
#include <stdio.h>

// Function to calculate the average of an array
float calculateAverage(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i]; // Adding all elements
    }
    return (float)sum / size; // Returning the average
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

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

    // Function call to calculate average
    printf("The average is: %.2f\n", calculateAverage(arr, n));

    return 0;
}
```

23. Write a simple program that passes an array to a function to calculate its sum.

```c
#include <stdio.h>

// Function to calculate sum of an array
int calculateSum(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i]; // Adding elements
    }
    return sum;
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

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

    // Function call to calculate sum
    printf("The sum of elements is: %d\n", calculateSum(arr, n));

    return 0;
}
```

24. How do you pass an array to a function in C? Write an example.

In C, arrays are passed to functions by passing the array **name** (which acts as a pointer to the first element). The function can then access and modify the elements.

```c
#include <stdio.h>

// Function to print an array
void printArray(int arr[], int size) {
    printf("Array elements are: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    // Passing array to function
    printArray(arr, size);

    return 0;
}
```

25.Write a program that takes three command-line arguments (length, breadth, and height) and uses a macro to calculate and print the volume of a rectangular box.(9)

```c
#include <stdio.h>
#include <stdlib.h>  // For atoi()

// Macro to calculate the volume of a rectangular box
#define VOLUME(l, b, h) ((l) * (b) * (h))

int main(int argc, char *argv[]) {
    // Check if exactly 4 arguments are provided (program name + 3 numbers)
    if (argc != 4) {
        printf("Usage: %s <length> <breadth> <height>\n", argv[0]);
        return 1;
    }

    // Convert command-line arguments to integers
    int length = atoi(argv[1]);
    int breadth = atoi(argv[2]);
    int height = atoi(argv[3]);

    // Calculate volume using the macro
    int volume = VOLUME(length, breadth, height);

    // Print the volume
    printf("Volume of the rectangular box: %d\n", volume);

    return 0;
}
```

26. a) What are command-line arguments in C? Discuss their syntax and usage. (3 marks)

Definition:

Command-line arguments allow users to pass inputs to a C program when executing it from the terminal or command prompt. These arguments are given after the program name and are received in the `main()` function.

Syntax

```c
int main(int argc, char *argv[])
```

Usage:

- Used to provide input to a program without user interaction.
- Useful in automation, scripting, and handling runtime parameters.

b) Write a program to accept two numbers as command-line arguments and calculate their sum and product. (6 marks)

```c
#include <stdio.h>
#include <stdlib.h>  // For atoi()

int main(int argc, char *argv[]) {
    // Ensure exactly 3 arguments (program name + 2 numbers)
    if (argc != 3) {
        printf("Usage: %s <num1> <num2>\n", argv[0]);
        return 1;
    }

    // Convert arguments from string to integer
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);

    // Calculate sum and product
    int sum = num1 + num2;
    int product = num1 * num2;

    // Display results
    printf("Sum: %d\n", sum);
    printf("Product: %d\n", product);

    return 0;
}
```

27. a) Explain the advantages and disadvantages of macros in C with examples. (3 marks)

Advantages of Macros:

1. **Faster Execution:**
   - Macros replace code at compile time, avoiding function call overhead.
   - Example:

```c
#define SQUARE(x) ((x) * (x))
printf("%d", SQUARE(5)); // Output: 25
```

2. **Code Readability and Maintainability:**
   - Macros improve readability by using meaningful names instead of hardcoded values.
   - Example:

```
#define PI 3.14159
```

Disadvantages of Macros:

1. **No Type Checking:**
   - Macros don't check data types, leading to unintended behavior.
   - Example:

```
#define SQUARE(x) x*x
printf("%d", SQUARE(5+2)); // Incorrect result: 5+2*5+2 = 19
```

2. **Increased Code Size:**
   - Since macros expand inline, they increase the final compiled code size if used frequently.
3. **Difficult Debugging:**
   - Errors in macros are harder to trace because they don't appear in function call stacks.

b) Write a program that uses a macro to calculate the area of a rectangle and demonstrate its use. (6 marks)

```
#include <stdio.h>

// Macro to calculate area of a rectangle
#define AREA(length, breadth) ((length) * (breadth))

int main() {
    int length, breadth;

    // Input length and breadth
    printf("Enter length and breadth of the rectangle: ");
    scanf("%d %d", &length, &breadth);

    // Calculate and display area using the macro
    printf("Area of the rectangle: %d\n", AREA(length, breadth));

    return 0;
}
```

28. Differentiate between macros and functions in C. Illustrate with an example of a macro that calculates the cube of a number.

FEATURE	MACROS (`#DEFINE`)	FUNCTIONS
Execution	Faster (code is replaced at compile time).	Slightly slower (requires function call overhead).
Type Checking	No type checking.	Ensures type safety.
Debugging	Harder to debug.	Easier to debug.
Code Size	Increases code size (inline expansion).	Smaller due to reusable function calls.
Flexibility	Limited (no loops, recursion).	More powerful (supports loops, recursion).

```c
#include <stdio.h>

#define CUBE(x) ((x) * (x) * (x))

int main() {
    int num = 3;
    printf("Cube of %d is: %d\n", num, CUBE(num));
    return 0;
}
```

29. Explain command-line arguments in C and write a program to print all command-line arguments passed to the program.

Command-line arguments allow users to pass values to a program when executing it. They are accessed using:

```c
int main(int argc, char *argv[])
```

Eg

Eg

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Total Arguments: %d\n", argc);

    // Printing all command-line arguments
    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

30. What are macros in C? Write a program that defines a macro to calculate the square of a number.

Macros (`#define`) are preprocessor directives that replace a specific code pattern with a value or expression before compilation. They improve code readability and execution speed.

```c
#include <stdio.h>

// Macro to calculate square
#define SQUARE(x) ((x) * (x))

int main() {
    int num;

    // User input
    printf("Enter a number: ");
    scanf("%d", &num);

    // Using macro to calculate square
    printf("Square of %d is: %d\n", num, SQUARE(num));

    return 0;
}
```

31. Write a C program to:
a) Create a structure with fields: Name, Address, Date of birth. (2 Marks)

b) Read the above details for five students from the user and display the details. (7 Marks)

```c
#include <stdio.h>

// Define a structure for student details
struct Student {
    char name[50];
    char address[100];
    char dob[15];  // Format: DD-MM-YYYY
};

int main() {
    struct Student students[5]; // Array to store details of 5 students

    // Reading details from the user
    for (int i = 0; i < 5; i++) {
        printf("\nEnter details for Student %d:\n", i + 1);

        printf("Enter Name: ");
        scanf(" %[^\n]", students[i].name);  // Reads full name including spaces

        printf("Enter Address: ");
        scanf(" %[^\n]", students[i].address);

        printf("Enter Date of Birth (DD-MM-YYYY): ");
        scanf(" %[^\n]", students[i].dob);
    }

    // Displaying the details
    printf("\nStudent Details:\n");
    for (int i = 0; i < 5; i++) {
        printf("\nStudent %d:\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Address: %s\n", students[i].address);
        printf("Date of Birth: %s\n", students[i].dob);
    }

    return 0;
}
```

32. Read two inputs each representing the distances between two points in the Euclidean space, store these in structure variables and add the two distance values.

```c
#include <stdio.h>
// Define a structure to store distance
struct Distance {
    int feet;
    int inches;
};
// Function to add two distances
struct Distance addDistances(struct Distance d1, struct Distance d2) {
    struct Distance result;

    // Add feet and inches separately
    result.feet = d1.feet + d2.feet;
    result.inches = d1.inches + d2.inches;

    // Convert inches to feet if inches >= 12
    if (result.inches >= 12) {
        result.feet += result.inches / 12;
        result.inches = result.inches % 12;
    }

    return result;
}

int main() {
    struct Distance d1, d2, sum;

    // Input first distance
    printf("Enter first distance (feet inches): ");
    scanf("%d %d", &d1.feet, &d1.inches);

    // Input second distance
    printf("Enter second distance (feet inches): ");
    scanf("%d %d", &d2.feet, &d2.inches);

    // Add distances
    sum = addDistances(d1, d2);

    // Display the result
    printf("\nTotal Distance: %d feet %d inches\n", sum.feet, sum.inches);

    return 0;
}
```

33. Write a C program to : a) Create a structure containing the fields: Name, Price, Quantity, Total Amount. (4 Marks)
b) Use separate functions to read and print the data. (5 Marks)

```c
#include <stdio.h>
// Define a structure to store product details
struct Product {
    char name[50];
    float price;
    int quantity;
    float totalAmount;
};
// Function to read product details
void readProduct(struct Product *p) {
    printf("\nEnter Product Name: ");
    scanf(" %[^\n]", p->name); // Reads input including spaces

    printf("Enter Price: ");
    scanf("%f", &p->price);

    printf("Enter Quantity: ");
    scanf("%d", &p->quantity);

    // Calculate total amount
    p->totalAmount = p->price * p->quantity;
}
// Function to display product details
void printProduct(struct Product p) {
    printf("\nProduct Details:\n");
    printf("Name: %s\n", p.name);
    printf("Price: %.2f\n", p.price);
    printf("Quantity: %d\n", p.quantity);
    printf("Total Amount: %.2f\n", p.totalAmount);
}

int main() {
    struct Product prod;

    // Read product details
    readProduct(&prod);

    // Display product details
    printProduct(prod);

    return 0;
}
```

34. Define a structure Book with members title, author, and price. Write a

program to define a structure variable, assign values to its members, and display the information.

```c
#include <stdio.h>

// Define the structure Book
struct Book {
    char title[100];
    char author[100];
    float price;
};

int main() {
    // Declare a structure variable
    struct Book myBook;

    // Assign values to structure members
    printf("Enter Book Title: ");
    scanf(" %[^\n]", myBook.title);

    printf("Enter Author: ");
    scanf(" %[^\n]", myBook.author);

    printf("Enter Price: ");
    scanf("%f", &myBook.price);

    // Display book information
    printf("\nBook Details:\n");
    printf("Title: %s\n", myBook.title);
    printf("Author: %s\n", myBook.author);
    printf("Price: %.2f\n", myBook.price);

    return 0;
}
```

35. What is a structure in C? Define a structure Student with members name, roll_no, and marks, and demonstrate how to access its members

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

```c
#include <stdio.h>

// Define structure Student
struct Student {
    char name[50];
    int roll_no;
    float marks;
};

int main() {
    // Declare a structure variable
    struct Student s1;

    // Assign values to structure members
    printf("Enter Student Name: ");
    scanf(" %[^\n]", s1.name);

    printf("Enter Roll Number: ");
    scanf("%d", &s1.roll_no);

    printf("Enter Marks: ");
    scanf("%f", &s1.marks);

    // Display student information
    printf("\nStudent Details:\n");
    printf("Name: %s\n", s1.name);
    printf("Roll Number: %d\n", s1.roll_no);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

36. How does an array differ from a structure?

FEATURE	ARRAY	STRUCTURE
Definition	Collection of elements of the same data type.	Collection of variables of different data types.
Data Type	Only one data type allowed (e.g., `int arr[10]`).	Can contain multiple data types (e.g., `char`, `int`, `float`).
Memory Allocation	Contiguous memory allocation.	Memory is allocated based on the structure's definition.
Access Method	Accessed using index notation (`arr[i]`).	Accessed using dot (`.`) operator (`student.name`).
Usage	Best for storing large amounts of similar data.	Best for grouping related attributes of an entity.

37.Declare a union containing five string variables: Name, House Name, City Name, State, and Pin Code. Each string should have a length of C_SIZE (a user-defined constant). Then, write a program to read and display the address of a person using a variable of this union.

```c
#include <stdio.h>
#include <string.h>
#define C_SIZE 50  // User-defined constant for string size

// Define a union to store address details
union Address {
    char name[C_SIZE];
    char houseName[C_SIZE];
    char cityName[C_SIZE];
    char state[C_SIZE];
    char pinCode[C_SIZE];
};

int main() {
    union Address addr;  // Declare a union variable

    // Read and display each field separately (since union stores one value at a time)
    printf("Enter Name: ");
    scanf(" %[^\n]", addr.name);
    printf("Stored Name: %s\n", addr.name);

    printf("\nEnter House Name: ");
    scanf(" %[^\n]", addr.houseName);
    printf("Stored House Name: %s\n", addr.houseName);

    printf("\nEnter City Name: ");
    scanf(" %[^\n]", addr.cityName);
    printf("Stored City Name: %s\n", addr.cityName);

    printf("\nEnter State: ");
    scanf(" %[^\n]", addr.state);
    printf("Stored State: %s\n", addr.state);

    printf("\nEnter Pin Code: ");
    scanf(" %[^\n]", addr.pinCode);
    printf("Stored Pin Code: %s\n", addr.pinCode);

    // Only the last assigned value remains valid in the union
    printf("\nFinal Value Stored in Union: %s\n", addr.pinCode);

    return 0;
}
```

38. Using a structure, write a program to read and print the data of n employees, including their Name, Employee ID, and Salary.

```c
#include <stdio.h>\
// Define a structure to store employee details
struct Employee {
    char name[50];
    int empID;
    float salary;
};
int main() {
    int n, i;

    // Get the number of employees
    printf("Enter the number of employees: ");
    scanf("%d", &n);

    // Declare an array of structures
    struct Employee employees[n];

    // Read employee details
    for (i = 0; i < n; i++) {
        printf("\nEnter details for Employee %d:\n", i + 1);

        printf("Enter Name: ");
        scanf(" %[^\n]", employees[i].name);

        printf("Enter Employee ID: ");
        scanf("%d", &employees[i].empID);

        printf("Enter Salary: ");
        scanf("%f", &employees[i].salary);
    }

    // Display employee details
    printf("\nEmployee Details:\n");
    for (i = 0; i < n; i++) {
        printf("\nEmployee %d:\n", i + 1);
        printf("Name: %s\n", employees[i].name);
        printf("Employee ID: %d\n", employees[i].empID);
        printf("Salary: %.2f\n", employees[i].salary);
    }

    return 0;
}
```

39. Declare a structure named Student to store the details (roll number, name, mark_for_C) of a student. Then, write a program in C to find the average mark obtained by the students in a class for the subject Programming in C (using the field mark_for_C). Use an array of structures to store the required data.

```c
#include <stdio.h>
// Define a structure to store student details
struct Student {
    int roll_no;
    char name[50];
    float mark_for_C;
};
int main() {
    int n, i;
    float total = 0, average;

    // Get the number of students
    printf("Enter the number of students: ");
    scanf("%d", &n);

    // Declare an array of structures
    struct Student students[n];

    // Read student details
    for (i = 0; i < n; i++) {
        printf("\nEnter details for Student %d:\n", i + 1);

        printf("Enter Roll Number: ");
        scanf("%d", &students[i].roll_no);

        printf("Enter Name: ");
        scanf(" %[^\n]", students[i].name);

        printf("Enter Mark for Programming in C: ");
        scanf("%f", &students[i].mark_for_C);

        // Add marks to total
        total += students[i].mark_for_C;
    }

    // Calculate average
    average = total / n;

    // Display student details
    printf("\nStudent Details:\n");
    for (i = 0; i < n; i++) {
        printf("\nStudent %d:\n", i + 1);
        printf("Roll Number: %d\n", students[i].roll_no);
        printf("Name: %s\n", students[i].name);
        printf("Mark for Programming in C: %.2f\n", students[i].mark_for_C);
    }

    // Display average marks
    printf("\nAverage Mark for Programming in C: %.2f\n", average);

    return 0;
}
```

40. What is an array of structures? Define an array of structures Student with members name, roll_no, and marks for 5 students, and demonstrate how to access its elements.

An **array of structures** is a collection of structure variables stored in a contiguous memory block. Each element in the array represents a structure instance, allowing multiple records to be managed efficiently.

```c
#include <stdio.h>

// Define a structure for student details
struct Student {
    char name[50];
    int roll_no;
    float marks;
};

int main() {
    // Declare an array of structures for 5 students
    struct Student students[5];

    // Input student details
    for (int i = 0; i < 5; i++) {
        printf("\nEnter details for Student %d:\n", i + 1);
        printf("Enter Name: ");
        scanf(" %[^\n]", students[i].name);
        printf("Enter Roll Number: ");
        scanf("%d", &students[i].roll_no);
        printf("Enter Marks: ");
        scanf("%f", &students[i].marks);
    }

    // Display student details
    printf("\nStudent Details:\n");
    for (int i = 0; i < 5; i++) {
        printf("\nStudent %d:\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].roll_no);
        printf("Marks: %.2f\n", students[i].marks);
    }

    return 0;
}
```

41. List out the advantages and disadvantages of union over structure in C

ADVANTAGES OF UNION	DISADVANTAGES OF UNION
Memory Efficient: Uses shared memory for all members, saving space.	**One Member at a Time**: Only one field holds a value at a time.
Flexible for Different Data Types: Useful when storing different data types in the same memory location.	**Data Loss**: Assigning a new value to one member overwrites the previous value.
Useful in Embedded Systems: Often used to handle memory-constrained applications.	**Difficult to Debug**: Since only the last written member is valid, debugging can be tricky.

42. Differentiate between a structure and a union in C with an example.

STRUCTURE	UNION
Allocates **separate memory** for each member.	**Shares memory** among all members.
All members can be accessed **simultaneously**.	Only **one member** can hold a value at a time.
Uses **more memory** than a union.	**Memory-efficient** since only one member is stored at a time.
Example: Used for **storing multiple attributes** of an entity.	Example: Used for **storing different data types in the same memory location**.

```c
#include <stdio.h>

// Define a structure
struct Student {
    char name[50];
    int roll_no;
    float marks;
};

// Define a union
union Data {
    int i;
    float f;
    char str[20];
};

int main() {
    struct Student s1 = {"Alice", 101, 85.5};
    union Data u1;

    // Assigning values to structure members
    printf("Structure Example:\n");
    printf("Name: %s\n", s1.name);
    printf("Roll Number: %d\n", s1.roll_no);
    printf("Marks: %.2f\n", s1.marks);

    // Assigning values to union members
    printf("\nUnion Example:\n");
    u1.i = 10;
    printf("Integer: %d\n", u1.i);
    u1.f = 12.5;
    printf("Float: %.2f\n", u1.f);  // Overwrites integer value
    return 0;
}
```

43. Write a note on storage classes in C, providing suitable examples.

Storage Classes in C

A **storage class** in C defines the scope, visibility, and lifetime of variables. There are four main storage classes in C:

1. **Automatic** (`auto`)
2. **External** (`extern`)
3. **Static** (`static`)
4. **Register** (`register`)

1. Automatic Storage Class (`auto`)

- Default storage class for local variables.
- The variable is created when the function is called and destroyed when the function exits.
- Accessible **only within the function** where it is declared.

eg

```c
#include <stdio.h>

void function() {
    auto int num = 10; // Automatic variable
    printf("Auto variable num = %d\n", num);
}

int main() {
    function();
    return 0;
}
```

2. External Storage Class (`extern`)

- Used for **global variables** that need to be accessed across multiple files.
- Declared outside all functions.
- Holds its value throughout the execution of the program.

```c
#include <stdio.h>

int globalVar = 50;  // Global variable

void printValue() {
    extern int globalVar; // Accessing external variable
    printf("Global variable: %d\n", globalVar);
}

int main() {
    printValue();
    return 0;
}
```

3. Static Storage Class (`static`)

- Retains its **value between function calls**.
- Default value is **zero (0) for numeric types**.
- Local static variables are **not destroyed** after function execution.

eg

```c
#include <stdio.h>

void counter() {
    static int count = 0; // Static variable
    count++;
    printf("Count = %d\n", count);
}

int main() {
    counter();
    counter();
    counter();
    return 0;
}
```

4. Register Storage Class (`register`)

- Stores the variable in the **CPU register** for faster access.
- Used for high-speed computations.
- Cannot get the address (`&`) of a register variable.

eg

```c
#include <stdio.h>

int main() {
    register int x = 10; // Register variable
    printf("Register variable x = %d\n", x);
    return 0;
}
```

44. a) Explain the automatic and static storage classes in C. Discuss the differences between them with the help of examples. (5 marks)

repeat

FEATURE	AUTO (AUTOMATIC)	STATIC (STATIC)
Scope	Local to function	Local to function
Lifetime	Created & destroyed each function call	Retains value throughout program execution
Initialization	Each function call	Only once
Default Value	Garbage (uninitialized)	0 (if not explicitly initialized)
Usage	Temporary calculations	Counters, loggers

b) Write a program to demonstrate the use of the static storage class by keeping track of the number of function calls. (4 marks)

```c
#include <stdio.h>

// Function to count function calls
void countCalls() {
    static int count = 0; // Static variable to retain count
    count++;
    printf("Function called %d times\n", count);
}

int main() {
    countCalls();
    countCalls();
    countCalls();
    return 0;
}
```

45. Explain the various C storage classes and provide examples for each.

Repeat

46. What is the register storage class in C? Explain its use with an example.

The **register storage class** in C is used to store variables in the **CPU registers** instead of RAM to improve **access speed**.

Characteristics:

- Stored in **CPU registers** for **faster access**.
- Used for **frequently accessed variables**, like loop counters.
- **Cannot use `&` (address-of operator)** to get the memory address.
- If registers are not available, it behaves like an **automatic (`auto`) variable**.

```c
#include <stdio.h>

int main() {
    register int i; // Register variable

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

    return 0;
}
```

47. What is the external storage class in C? Explain how it is used with an example.

The **external (`extern`) storage class** is used for **global variables** that can be accessed across multiple files.

Characteristics:

- The variable is **declared globally** outside all functions.
- Can be **accessed and modified by multiple functions**.
- Lifetime: **Till the program terminates.**
- Default value: **0 (if not initialized)**.

```c
#include <stdio.h>

int globalVar = 50; // External variable

void display() {
    extern int globalVar; // Accessing external variable
    printf("Global variable: %d\n", globalVar);
}

int main() {
    display();
    return 0;
}
```

48. What is the automatic storage class in C? Explain its characteristics with an example.

The **automatic** (`auto`) **storage class** is the **default storage class** for **local variables** in C.

Characteristics:

- The variable is created when a function is called and destroyed when it exits.
- Stored in **RAM (not CPU registers)**.
- Scope: **Limited to the function** where it is declared.
- Default value: **Garbage (uninitialized)**.

```c
#include <stdio.h>

void function() {
    auto int num = 10; // Automatic variable
    printf("Auto variable num = %d\n", num);
}

int main() {
    function();
    function(); // The variable num is reinitialized every time
    return 0;
}
```