36. A teacher wants to analyze the marks of students in a class. The teacher needs a program that: ● Accepts marks of N students. ● Finds the highest and lowest marks. ● Sorts the marks in ascending order. ● Searches for a specific student's mark using Binary Search.

and

37. Write a C program to implement the above functionalities using arrays, functions, and Binary Search.

```c
#include <stdio.h>
#include <limits.h> // For INT_MIN and INT_MAX
// Function to print the elements of an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
// Function to find the highest and lowest marks
void findMinMax(int arr[], int size, int *min_mark, int *max_mark) {
    *min_mark = INT_MAX; // Initialize min with a very large value
    *max_mark = INT_MIN; // Initialize max with a very small value
    for (int i = 0; i < size; i++) {
        if (arr[i] < *min_mark) {
            *min_mark = arr[i];
        }
        if (arr[i] > *max_mark) {
            *max_mark = arr[i];
        }
    }
}
```

```
// Function to sort an array in ascending order using Bubble Sort
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
// Function to search for a specific mark in a sorted array using Binary Search
// Returns the index if found, otherwise -1
int binarySearch(int arr[], int size, int key) {
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = low + (high - low) / 2; // Calculate middle index

        if (arr[mid] == key) {
            return mid; // Key found
        }
        else if (arr[mid] < key) {
            low = mid + 1; // Search in the right half
        }
        else {
            high = mid - 1; // Search in the left half
```

```
        }
    }
    return -1; // Key not found
}
int main() {
    int N;
    int marks[100]; // Array to store marks
    int min_mark, max_mark;
    int search_mark;
    int found_index;
    // --- 1. Accept marks of N students ---
    printf("Enter the number of students (N, max 100): ");
    scanf("%d", &N);
    if (N <= 0 || N > 100) {
        printf("Error: N must be between 1 and 100.\n");
        return 1;
    }
    printf("Enter marks for %d students (integers):\n", N);
    for (int i = 0; i < N; i++) {
        printf("Mark for student %d: ", i + 1);
        scanf("%d", &marks[i]);
    }
    printf("\nOriginal Marks: ");
    printArray(marks, N);
    // --- 2. Find the highest and lowest marks ---
    findMinMax(marks, N, &min_mark, &max_mark);
    printf("Highest Mark: %d\n", max_mark);
    printf("Lowest Mark: %d\n", min_mark);
    // --- 3. Sort the marks in ascending order ---
```

```
    printf("\nSorting marks in ascending order...\n");
    bubbleSort(marks, N);
    printf("Sorted Marks: ");
    printArray(marks, N);
    // --- 4. Search for a specific student's mark using Binary Search ---
    printf("\nEnter a mark to search for: ");
    scanf("%d", &search_mark);
    // Binary search requires the array to be sorted
    found_index = binarySearch(marks, N, search_mark);
    if (found_index != -1) {
        printf("Mark %d found at sorted index %d (position %d).\n",
               search_mark, found_index, found_index + 1);
    } else {
        printf("Mark %d not found in the list.\n", search_mark);
    }
    return 0;
}
```

38. What is a structure in C? How is it different from an array? Explain with an example.

What is a Structure in C?

A structure (often shortened to struct) in C is a **user-defined data type** that allows you to combine items of **different data types** under a single name. Think of it as a blueprint or a template for creating complex data records. Each item within a structure is called a **member** (or field).

Purpose: Structures are particularly useful when you need to represent a real-world entity that has multiple, related attributes of varying data types. For example, a "Student" might have a name (string), a roll number (integer), and marks (float). A struct lets you bundle all these pieces of information together logically.

Declaration Syntax:

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

Creating a Structure Variable:

```
struct structure_name variable_name;
```

Accessing Members: You access members of a structure using the **dot operator (.)** for structure variables or the **arrow operator (->)** for pointers to structures.

How is a Structure Different from an Array?

While both arrays and structures are used to group data, their fundamental differences lie in the **type of data they can hold** and **how their elements are accessed**.

Feature	Structure (struct)	Array ([])
Data Types	Can group different (heterogeneous) data types.	Can group only the same (homogeneous) data type.
Element Identification	Members are identified by names (e.g., student.name).	Elements are identified by numerical indices (e.g., scores[0]).
Access Operator	Uses dot operator (.) or arrow operator (->)	Uses square brackets ([])
Conceptual View	A record, a blueprint, a collection of related attributes.	An ordered list, a sequence, a collection of similar items.
Memory Layout	Members are stored sequentially, but padding can occur between members for alignment purposes.	Elements are stored strictly contiguously in memory.
Primary Use	To represent a single, complex entity with varied attributes.	To store a collection of similar items.

Initialization	Members initialized using brace-enclosed list or member-wise assignment.	Elements initialized using brace-enclosed list.

Example: struct Student vs. int marks[]

Let's illustrate the differences using an example of storing student data.

```c
#include <stdio.h> // For input/output functions
#include <string.h> // For string manipulation (strcpy)
// --- Example of a Structure ---
// Define a structure named 'Student'
struct Student {
    char name[50];    // Member 1: A character array (string) for name
    int roll_number;  // Member 2: An integer for roll number
    float marks;      // Member 3: A float for marks
};
int main() {
    printf("--- Demonstrating Structures ---\n");
    // 1. Declare a variable of type 'struct Student'
    struct Student student1;
    // 2. Initialize members of the structure variable
    // For char arrays (strings), use strcpy
    strcpy(student1.name, "Alice Smith");
    student1.roll_number = 101;
    student1.marks = 85.5;

    // 3. Access and print members of the structure
    printf("Student Name: %s\n", student1.name);
    printf("Roll Number: %d\n", student1.roll_number);
    printf("Marks: %.2f\n", student1.marks);
    printf("\n--- Demonstrating Arrays ---\n");
```

```
    // --- Example of an Array ---
    // Declare and initialize an array of integers (e.g., marks of students in
a single subject)
    int studentMarksArray[3] = {75, 88, 92};
    // Access and print elements of the array
    printf("Student marks in an array:\n");
    printf("Mark 1 (index 0): %d\n", studentMarksArray[0]);
    printf("Mark 2 (index 1): %d\n", studentMarksArray[1]);
    printf("Mark 3 (index 2): %d\n", studentMarksArray[2]);
    printf("\n--- Array of Structures ---\n");
    // You can also have an array of structures, which combines both concepts
    struct Student class_students[2];
    strcpy(class_students[0].name, "Bob Johnson");
    class_students[0].roll_number = 102;
    class_students[0].marks = 78.0;
    strcpy(class_students[1].name, "Charlie Brown");
    class_students[1].roll_number = 103;
    class_students[1].marks = 91.2;
    printf("Details of students in class_students array:\n");
    for (int i = 0; i < 2; i++) {
        printf("Student %d: Name=%s, Roll=%d, Marks=%.2f\n",
               i + 1, class_students[i].name, class_students[i].roll_number,
class_students[i].marks);
    }
    return 0;
}
```

Explanation of the Example:

1. **struct Student (Structure Example):**
    - We define a struct Student that has three members: name (a char array for strings), roll_number (an int), and marks (a float). Notice how they are all different data types.

- We then declare a variable student1 of type struct Student.
- We initialize its members individually using the **dot operator (.)**: student1.name, student1.roll_number, student1.marks.
- We access and print these members also using the dot operator.
- This struct effectively creates a single "record" for one student.

2. **int studentMarksArray[] (Array Example):**
   - We declare an int array studentMarksArray to store marks.
   - All elements in this array *must* be of type int. We cannot store a student's name or roll number directly within this int array.
   - We access elements using their **numerical index**: studentMarksArray[0], studentMarksArray[1], etc.
3. **struct Student class_students[2] (Array of Structures):**
   - This demonstrates that you can even combine both concepts: create an array where each element is a struct Student. This allows you to store records for multiple students. Each class_students[i] is a complete Student record, which you then access its members using the dot operator (e.g., class_students[i].name).

39. Differentiate between struct and union in C with an example.

Differentiating struct and union in C

1. struct (Structure)

- **Definition:** A struct is a user-defined data type that allows you to combine items of **different data types** under a single name. Each member within a struct is stored in its **own, distinct memory location**.
- **Memory Allocation:** The compiler allocates enough memory to store **all** the members of the structure. The total size of the structure is the sum of the sizes of its members (plus potential padding added by the compiler for alignment).
- **Simultaneous Access:** You can **access and use all members simultaneously** at any given time.
- **Purpose:** Ideal for grouping related data items that logically exist together and are all relevant at the same time (e.g., a student record with name, roll number, and marks).

2. union (Union)

- **Definition:** A union is also a user-defined data type that allows you to combine items of **different data types** under a single name. However, all members of a union share the **same memory location**.

- **Memory Allocation:** The compiler allocates memory equal to the size of its **largest** member. This single memory block is then used by *all* members, but **only one member can hold a value at any given time**.
- **Simultaneous Access:** You can **only access one member at a time** – specifically, the one that was most recently written to. If you write to one member and then try to read from another, you will get corrupted or undefined data, as the memory content has been overwritten.
- **Purpose:** Primarily used for memory optimization when you know that only one of a set of data types will be relevant at any given moment (e.g., a data packet that can either be an integer, a float, or a string, but never all at once). It provides a way to overlay different data types in the same memory space.

Feature	struct (Structure)	union (Union)
Keyword	struct	union
Memory Allocation	Allocates memory for all members. Total size is sum of member sizes (plus padding).	Allocates memory for the largest member only. Total size is size of largest member.
Member Storage	Each member has its own unique memory location.	All members share the same memory location.
Simultaneous Access	All members can be accessed simultaneously.	Only one member can be actively used/accessed at any given time (the last one written to).
Data Integrity	Writing to one member does not affect others.	Writing to one member overwrites the data of other members.
Purpose	Grouping different, logically distinct attributes.	Memory optimization when only one member's value is relevant at a time.

Example: struct Data vs. union Data

Let's use an example where we try to store different types of data (int, float, char) in both a struct and a union to see the memory and access behavior.

```
#include <stdio.h> // For input/output functions

// Define a structure named 'MyStruct'

struct MyStruct {

    int   i;    // Integer member

    float f;    // Float member
```

```
    char  c;    // Character member
};
// Define a union named 'MyUnion'
union MyUnion {
    int   i;    // Integer member
    float f;    // Float member
    char  c;    // Character member
};
int main() {
    printf("--- Differentiating Struct vs Union ---\n");
    // --- STRUCT DEMONSTRATION ---
    printf("\n--- STRUCT Example ---\n");
    struct MyStruct s; // Declare a variable of type MyStruct
    // Get sizes of members and the struct itself
    printf("Size of int in struct: %zu bytes\n", sizeof(s.i));
    printf("Size of float in struct: %zu bytes\n", sizeof(s.f));
    printf("Size of char in struct: %zu bytes\n", sizeof(s.c));
    printf("Size of MyStruct (total memory allocated): %zu bytes\n",
sizeof(s));
    // Store values in struct members
    s.i = 10;
    s.f = 20.5f;
    s.c = 'A';
    // Access and print all members of the struct
    printf("MyStruct values: i = %d, f = %.1f, c = %c\n", s.i, s.f, s.c);
    // All values are preserved and can be accessed simultaneously.
    // --- UNION DEMONSTRATION ---
    printf("\n--- UNION Example ---\n");
    union MyUnion u; // Declare a variable of type MyUnion
    // Get sizes of members and the union itself
```

```
    printf("Size of int in union: %zu bytes\n", sizeof(u.i));
    printf("Size of float in union: %zu bytes\n", sizeof(u.f));
    printf("Size of char in union: %zu bytes\n", sizeof(u.c));
    // Notice that the size of the union is equal to the size of its largest
member (float or int, typically 4 bytes)
    printf("Size of MyUnion (total memory allocated): %zu bytes\n", sizeof(u));
    // Store value in 'i' member of union
    u.i = 123;
    printf("\nAfter assigning u.i = %d:\n", u.i);
    printf("u.i: %d\n", u.i);
    // Other members will contain garbage or overwritten bits as they share the
same memory
    printf("u.f: %.1f (garbage - memory was formatted for int)\n", u.f);
    printf("u.c: %c (garbage - memory was formatted for int)\n", u.c);
    // Now, store value in 'f' member of union
    u.f = 45.6f;
    printf("\nAfter assigning u.f = %.1f:\n", u.f);
    printf("u.f: %.1f\n", u.f);
    // 'i' and 'c' are now corrupted because 'f' overwrote the shared memory
    printf("u.i: %d (garbage - memory was formatted for float)\n", u.i);
    printf("u.c: %c (garbage - memory was formatted for float)\n", u.c);
    // Finally, store value in 'c' member of union
    u.c = 'Z';
    printf("\nAfter assigning u.c = %c:\n", u.c);
    printf("u.c: %c\n", u.c);
    // 'i' and 'f' are now corrupted because 'c' overwrote the shared memory
    printf("u.i: %d (garbage - memory was formatted for char)\n", u.i);
    printf("u.f: %.1f (garbage - memory was formatted for char)\n", u.f);
    return 0;
}
```

Explanation of the Example Output:

When you run this program:

1. **Struct Output:**
    - You'll see that sizeof(MyStruct) is typically the sum of its members' sizes (e.g., 4 + 4 + 1 = 9 bytes, but due to padding, it might be 12 bytes on a 4-byte aligned system).
    - When you store 10 in s.i, 20.5f in s.f, and 'A' in s.c, you can then print all three values, and they will be correct. This confirms that each member has its own dedicated memory space.
2. **Union Output:**
    - You'll see that sizeof(MyUnion) is typically 4 bytes (the size of int or float, whichever is larger). This shows that all members share the same 4-byte memory location.
    - When you assign u.i = 123; and then try to print u.f or u.c, you will get garbage values. The memory now contains the bit pattern for the integer 123, which doesn't make sense when interpreted as a float or a character.
    - When you then assign u.f = 45.6f;, the previous integer value 123 is overwritten. Now u.i and u.c will show garbage.
    - The same happens when you assign u.c = 'Z';.

40. What are the different modes in which files can be opened in C? Explain with examples.

In C, when you work with files, you need to specify how you intend to interact with the file (e.g., read from it, write to it, append to it). This is done by providing a **file mode** string to the fopen() function. The mode determines the operations allowed on the file and its initial state.

Here are the different modes in which files can be opened in C, along with examples:

To open a file, you use the fopen() function, which returns a file pointer (FILE *). If fopen() fails (e.g., file not found in read mode), it returns NULL.

```
FILE *fptr;

fptr = fopen("filename.txt", "mode");
```

File Opening Modes:

1. **"r" (Read Mode)**
    - **Purpose:** Opens an existing file for reading.
    - **Behavior:**

- The file pointer is placed at the **beginning** of the file.
- If the file **does not exist**, fopen() returns NULL.

- **Example:** Reading content from a file.

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    char buffer[100];
    // Create a dummy file for demonstration if it doesn't exist
    fptr = fopen("example_read.txt", "w");
    if (fptr != NULL) {
        fprintf(fptr, "Hello, C File Handling!\n");
        fprintf(fptr, "This is a test line.\n");
        fclose(fptr);
    }
    // Open the file in read mode
    fptr = fopen("example_read.txt", "r");
    if (fptr == NULL) {
        printf("Error: Could not open file for reading.\n");
    } else {
        printf("--- Reading from 'example_read.txt' ---\n");
        while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
            printf("%s", buffer);
        }
        fclose(fptr);
        printf("File closed.\n");
    }
    return 0;
}
```

2. **"w" (Write Mode)**
    - **Purpose:** Opens a file for writing.
    - **Behavior:**
        - If the file **exists**, its contents are **truncated** (deleted), and the file is treated as a new, empty file.
        - If the file **does not exist**, a new file with the specified name is **created**.
        - The file pointer is placed at the **beginning** of the file.
    - **Example:** Writing new content to a file.

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    // Open the file in write mode
    fptr = fopen("example_write.txt", "w");
    if (fptr == NULL) {
        printf("Error: Could not open file for writing.\n");
    } else {
        printf("--- Writing to 'example_write.txt' ---\n");
        fprintf(fptr, "This is the first line.\n");
        fprintf(fptr, "This line overwrites any previous content.\n");
        fclose(fptr);
        printf("Content written and file closed.\n");
    }
    return 0;
}
```

3. **"a" (Append Mode)**
    - **Purpose:** Opens a file for appending (adding content to the end).
    - **Behavior:**
        - If the file **exists**, the file pointer is placed at the **end** of the file. New data will be written after the existing content.

- If the file **does not exist**, a new file with the specified name is **created**.

- **Example:** Adding new lines to an existing file.

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    // Open the file in append mode
    fptr = fopen("example_append.txt", "a");
    if (fptr == NULL) {
        printf("Error: Could not open file for appending.\n");
    } else {
        printf("--- Appending to 'example_append.txt' ---\n");
        fprintf(fptr, "This line is appended.\n");
        fprintf(fptr, "Another line added at the end.\n");
        fclose(fptr);
        printf("Content appended and file closed.\n");
    }
    return 0;
}
```

4. **"r+" (Read and Write Mode)**
    - **Purpose:** Opens an existing file for both reading and writing.
    - **Behavior:**
        - The file must **exist**. If it doesn't, fopen() returns NULL.
        - The file pointer is placed at the **beginning** of the file.
        - Allows both reading from and writing to the file. Writing will overwrite existing content from the current pointer position.
    - **Example:** Reading and then modifying content in place.

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    // Create a dummy file for demonstration
    fptr = fopen("example_rplus.txt", "w");
    if (fptr != NULL) {
        fprintf(fptr, "Original line 1.\nOriginal line 2.\n");
        fclose(fptr);
    }
    // Open the file in r+ mode
    fptr = fopen("example_rplus.txt", "r+");
    if (fptr == NULL) {
        printf("Error: Could not open file for r+ mode.\n");
    } else {
        printf("--- Using 'r+' mode on 'example_rplus.txt' ---\n");
        char buffer[100];
        fgets(buffer, sizeof(buffer), fptr); // Read first line
        printf("Read: %s", buffer);
        fseek(fptr, 0, SEEK_SET); // Go back to the beginning of the file
        fprintf(fptr, "NEW first line."); // Overwrite part of the first line
        fclose(fptr);
        printf("File modified and closed. Check file content.\n");
    }
    return 0;
}
```

5. **"w+" (Write and Read Mode)**
    - **Purpose:** Opens a file for both writing and reading.
    - **Behavior:**
        - If the file **exists**, its contents are **truncated** (deleted).

- If the file **does not exist**, a new file is **created**.
- The file pointer is placed at the **beginning** of the file.
- Allows both reading and writing.

- **Example:** Create/truncate, write, then read from the same file.

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    // Open the file in w+ mode (will truncate if exists, create if not)
    fptr = fopen("example_wplus.txt", "w+");
    if (fptr == NULL) {
        printf("Error: Could not open file for w+ mode.\n");
    } else {
        printf("--- Using 'w+' mode on 'example_wplus.txt' ---\n");
        fprintf(fptr, "Content written with w+.\n"); // Write some content
        fseek(fptr, 0, SEEK_SET); // Move file pointer to the beginning to read
        char buffer[100];
        fgets(buffer, sizeof(buffer), fptr); // Read the content just written
        printf("Read: %s", buffer);
        fclose(fptr);
        printf("File closed.\n");
    }
    return 0;
}
```

6. **"a+" (Append and Read Mode)**
   - **Purpose:** Opens a file for both appending and reading.
   - **Behavior:**
     - If the file **exists**, the file pointer is initially at the **end** for writing.
     - If the file **does not exist**, a new file is **created**.

- You can read from anywhere in the file (by moving the file pointer with fseek), but all *writes* will always occur at the end of the file.

- **Example:** Read existing content, then append new content.

```
#include <stdio.h>
int main() {
    FILE *fptr;   // Create a dummy file for demonstration
    fptr = fopen("example_aplus.txt", "w");
    if (fptr != NULL) {
        fprintf(fptr, "Initial content.\n");
        fclose(fptr);
    }
    fptr = fopen("example_aplus.txt", "a+");    // Open the file in a+ mode
    if (fptr == NULL) {
        printf("Error: Could not open file for a+ mode.\n");
    } else {
        printf("--- Using 'a+' mode on 'example_aplus.txt' ---\n");
        char buffer[100];
        fseek(fptr, 0, SEEK_SET); // Move to the beginning to read existing
content
        printf("Reading existing content:\n");
        while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
            printf("%s", buffer);
        }
     // File pointer is now at the end (after reading all content or initially)
        fprintf(fptr, "This is new appended content.\n");
        printf("New content appended.\n");
        fclose(fptr);
        printf("File closed.\n");
    }
    return 0;}
```

41. Write a C program to define a structure named Student with members: name, roll_no, and marks. Accept details of a student and display them

```c
#include <stdio.h>
#include <string.h>
// Define the structure named 'Student'
struct Student {
    char name[50];
    int roll_no;
    float marks;
};
int main() {
    struct Student student1;
    printf("--- Enter Student Details ---\n");
    printf("Enter student's Name: ");
    fgets(student1.name, sizeof(student1.name), stdin);
    student1.name[strcspn(student1.name, "\n")] = '\0'; // Remove newline
    printf("Enter student's Roll Number: ");
    scanf("%d", &student1.roll_no);
    printf("Enter student's Marks: ");
    scanf("%f", &student1.marks);
    printf("\n--- Student Details ---\n");
    printf("Name: %s\n", student1.name);
    printf("Roll Number: %d\n", student1.roll_no);
    printf("Marks: %.2f\n", student1.marks);
    return 0;
}
```

42. Write a C program to create a file and write a string into it.

```c
#include <stdio.h> // Required for file operations (fopen, fprintf, fclose)

int main() {

    FILE *fptr; // Declare a file pointer

    char file_name[] = "my_output_file.txt"; // Name of the file to create/write

    char content[] = "Hello from C programming! This string will be written to the file."; // String to write

    // Step 1: Open the file in write mode ("w")

    // If the file exists, its content will be truncated (deleted).

    // If it doesn't exist, a new file will be created.

    fptr = fopen(file_name, "w");

    // Step 2: Check if the file was opened successfully

    if (fptr == NULL) {

        printf("Error: Could not open/create the file '%s'.\n", file_name);

        return 1; // Indicate an error

    }

    // Step 3: Write the string into the file

    // fprintf behaves like printf but writes to the specified file stream

    fprintf(fptr, "%s\n", content); // Add a newline character for better readability in the file

    // Step 4: Close the file

    // It's crucial to close the file to save changes and release resources.

    fclose(fptr);

    printf("Successfully wrote the string to '%s'.\n", file_name);

    return 0; // Indicate successful execution

}
```

43. Write a C program to demonstrate how memory allocation differs in structures and unions by creating a structure and a union with the same data members.

```c
#include <stdio.h>
// Define a structure named 'MyStruct'
struct MyStruct {
    int   i;
    float f;
    char  c;
};
// Define a union named 'MyUnion' with the same members
union MyUnion {
    int   i;
    float f;
    char  c;
};
int main() {
    struct MyStruct s; // Declare a struct variable
    union MyUnion u;   // Declare a union variable
    printf("--- Memory Allocation Comparison: Struct vs. Union ---\n\n");
    // --- STRUCT Memory Allocation ---
    printf("--- STRUCT (MyStruct) ---\n");
    printf("Size of int member (s.i): %zu bytes\n", sizeof(s.i));
    printf("Size of float member (s.f): %zu bytes\n", sizeof(s.f));
    printf("Size of char member (s.c): %zu bytes\n", sizeof(s.c));
    printf("Total size of MyStruct: %zu bytes\n", sizeof(s));
    printf("Explanation: A struct allocates memory for ALL its members. The
total size is the sum of member sizes (plus potential padding for
alignment).\n\n");
```

```c
    // --- UNION Memory Allocation ---
    printf("--- UNION (MyUnion) ---\n");
    printf("Size of int member (u.i): %zu bytes\n", sizeof(u.i));
    printf("Size of float member (u.f): %zu bytes\n", sizeof(u.f));
    printf("Size of char member (u.c): %zu bytes\n", sizeof(u.c));
    printf("Total size of MyUnion: %zu bytes\n", sizeof(u));
    printf("Explanation: A union allocates memory equal to the size of its
LARGEST member. All members share this same memory location.\n\n");
    printf("--- Practical Demonstration of Union Overwriting ---\n");
    u.i = 123; // Assign value to integer member
    printf("After u.i = 123:\n");
    printf("  u.i: %d\n", u.i);
    printf("  u.f: %.2f (Garbage, as 'i' overwrote memory)\n", u.f);
    printf("  u.c: %c (Garbage, as 'i' overwrote memory)\n", u.c);
    u.f = 45.67f; // Assign value to float member (overwrites 'i')
    printf("\nAfter u.f = 45.67f:\n");
    printf("  u.f: %.2f\n", u.f);
    printf("  u.i: %d (Garbage, as 'f' overwrote memory)\n", u.i);
    printf("  u.c: %c (Garbage, as 'f' overwrote memory)\n", u.c);

    u.c = 'X'; // Assign value to char member (overwrites 'f')
    printf("\nAfter u.c = 'X':\n");
    printf("  u.c: %c\n", u.c);
    printf("  u.i: %d (Garbage, as 'c' overwrote memory)\n", u.i);
    printf("  u.f: %.2f (Garbage, as 'c' overwrote memory)\n", u.f);
    return 0;
}
```

44. What is the difference between writing to a file using "w" mode and "a" mode in fopen()? Explain with examples.

When working with files in C, the mode you choose with fopen() dictates how your program interacts with the file. The two modes "w" (write) and "a" (append) are both used for writing data, but they differ significantly in how they handle existing files and the initial position of the file pointer.

Difference Between "w" (Write Mode) and "a" (Append Mode)

1. "w" (Write Mode)

- **Purpose:** To open a file specifically for **writing** new content.
- **File Existence:**
    - If the file **exists**, its entire content is **truncated** (deleted). The file becomes empty, effectively starting fresh.
    - If the file **does not exist**, a new, empty file with the specified name is **created**.
- **File Pointer Position:** The file pointer is placed at the **beginning** of the file. Any data written will start from the very first byte.
- **Use Case:** When you want to create a new file or completely overwrite the contents of an existing file.

Example for "w" mode:

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    char filename[] = "example_w.txt";
    printf("--- Demonstrating 'w' (Write) mode ---\n");
    // First write: This will create the file and put content.
    // If it exists, it will be truncated first.
    fptr = fopen(filename, "w");
    if (fptr == NULL) {
        printf("Error opening %s for writing.\n", filename);
        return 1;
    }
    fprintf(fptr, "This is the first line written with 'w' mode.\n");
```

```
    fprintf(fptr, "Any previous content is now gone.\n");
    fclose(fptr);
    printf("Content written to '%s'. Check its content.\n", filename);
    // Second write with 'w' mode: This will OVERWRITE the file again.
    printf("\nWriting again to '%s' with 'w' mode (will truncate previous
content).\n", filename);
    fptr = fopen(filename, "w");
    if (fptr == NULL) {
        printf("Error opening %s for writing the second time.\n", filename);
        return 1;
    }
    fprintf(fptr, "This is the NEW content.\n");
    fprintf(fptr, "The old content has been completely replaced.\n");
    fclose(fptr);
    printf("New content written to '%s'. Check its content.\n", filename);
    return 0;
}
```

Expected Output in example_w.txt after running the program:

This is the NEW content.

The old content has been completely replaced.

(The first set of lines written will be gone.)

2. "a" (Append Mode)

- **Purpose:** To open a file specifically for **appending** new content to its end.
- **File Existence:**
    - If the file **exists**, its existing content is **preserved**.
    - If the file **does not exist**, a new, empty file with the specified name is **created**.
- **File Pointer Position:** The file pointer is placed at the **end** of the file. All new data written will be added after the last existing character.
- **Use Case:** When you want to add new data to an existing file without deleting its current contents (e.g., logging events, adding new entries to a list).

Example for "a" mode:

```c
#include <stdio.h>
int main() {
    FILE *fptr;
    char filename[] = "example_a.txt";
    printf("--- Demonstrating 'a' (Append) mode ---\n");
    // First write: This will create the file and put content.
    // If it exists, it will be truncated first (using 'w' for initial setup).
    // In a real scenario, you might just open directly with 'a'.
    fptr = fopen(filename, "w");
    if (fptr == NULL) {
        printf("Error setting up %s.\n", filename);
        return 1;
    }
    fprintf(fptr, "This is the initial content.\n");
    fclose(fptr);
    printf("Initial content written to '%s'.\n", filename);
    // Now, open with 'a' mode to append.
    printf("\nAppending to '%s' with 'a' mode.\n", filename);
    fptr = fopen(filename, "a");
    if (fptr == NULL) {
        printf("Error opening %s for appending.\n", filename);
        return 1;
    }
    fprintf(fptr, "This line is appended to the end.\n");
    fprintf(fptr, "Another line appended.\n");
    fclose(fptr);
    printf("Content appended to '%s'. Check its content.\n", filename);
    return 0;}
```

Expected Output in example_a.txt after running the program:

```
This is the initial content.
This line is appended to the end.
Another line appended.
```

45. A.Write the application/need of union in c. B.Write the advantages and disadvantages of union.

A. Application/Need of union in C

The primary application and need for a union in C revolves around **memory optimization** and creating **variant records** where different types of data might occupy the same conceptual space, but only one is active at any given time.

Here are specific scenarios where unions are typically used:

1. **Memory Optimization/Saving Space:**
    - This is the most common reason. If you have a data structure where certain fields are mutually exclusive (i.e., only one of them will hold a meaningful value at any given moment), using a union can save significant memory compared to a struct.
    - **Example:** A Message structure that can either hold an int command, a float value, or a char array (string) as payload. You know a message will never carry an int AND a float simultaneously.
2. **Implementing Variant Records/Tagged Unions:**
    - When dealing with data that can take on one of several different types, and you need a way to track which type is currently stored. A common pattern is to use a struct that contains a union for the varying data and a tag (an enum or int) to indicate which member of the union is currently valid.
    - **Example:** A generic Value type that could be an integer, a floating-point number, or a character, where the program dynamically determines its actual type.

```
enum DataType { INT_TYPE, FLOAT_TYPE, CHAR_TYPE };
struct GenericValue {
    enum DataType type; // Tag to indicate which union member is active
    union {
        int i_val;
```

```
        float f_val;
        char c_val;
    } data; // The union holding the actual value
};
// If type is INT_TYPE, use data.i_val. If FLOAT_TYPE, use data.f_val, etc.
```

3. **Type Punning (Reinterpreting Memory):**
    - Unions can be used to interpret the same block of memory as different data types. This is often done for low-level programming tasks like byte manipulation, network protocols, or hardware interaction.
    - **Example:** Checking the individual bytes of an int (to understand endianness).

```
union IntBytes {
    int val;
    char bytes[sizeof(int)];
};
// If you assign val = 0x12345678, you can then inspect bytes[0], bytes[1] etc.
```

    - *Caution:* While possible, type punning using unions can lead to undefined behavior or non-portable code if not handled carefully, especially regarding strict aliasing rules.

In summary, the need for unions arises when efficient memory usage is critical, and the problem inherently involves data that can assume one of several mutually exclusive forms.

B. Advantages and Disadvantages of union

Advantages:

1. **Memory Efficiency:** This is the primary advantage. By allowing different data types to share the same memory location, unions significantly reduce the memory footprint of a data structure, especially when dealing with large numbers of such structures.
2. **Flexible Type Handling (Variant Records):** They provide a clean way to handle situations where a variable or structure field might need to store different types of data at different times. When combined with an enum tag, they form robust "variant records."
3. **Type Punning (Low-Level Operations):** They enable advanced techniques like type punning, allowing you to access the same memory location as different data types. This is useful for byte-level manipulation, parsing binary data, and interacting with hardware.

Disadvantages:

1. **Lack of Type Safety:**

- This is the biggest drawback. The compiler does not track which member of a union is currently active or holds valid data. It's entirely up to the programmer to ensure they read from the member that was most recently written to.
- Reading from an inactive member leads to **undefined behavior** (often garbage values), which can be very difficult to debug.

2. **Only One Member Active:** Only one member of a union can hold a meaningful value at any given time. Assigning a value to one member overwrites any data previously stored in another member.
3. **Complexity and Debugging:** Managing unions, especially without a clear "tag" system, can make code harder to understand, maintain, and debug, as the meaning of the memory content changes based on the last write operation.
4. **Portability Issues (for Type Punning):** While type punning is an application, its reliance on how data types are represented in memory (e.g., endianness, padding) can lead to non-portable code that behaves differently on various systems.
5. **No Direct Member-Wise Initialization:** Unlike structures, you can only initialize the *first* member of a union directly in its declaration.

```
union MyUnion u = {10}; // Initializes u.i = 10

// union MyUnion u = {.f = 10.5f}; // Valid in C99/C11 designated initializers,
but o
```

46. Write a C program to define a structure for the students and then keep the records(Name, Roll No, Marks) of 5 students.

```
#include <stdio.h>

#include <string.h> // For fgets and strcspn

// Define the Student structure

struct Student {

    char name[50];

    int roll_no;

    float marks;

};

int main() {

    // Declare an array of 5 Student structures
```

```c
    struct Student students[5];
    int i; // Loop counter
    printf("--- Enter Details for 5 Students ---\n");
    // Loop to accept details for each of the 5 students
    for (i = 0; i < 5; i++) {
        printf("\nEnter details for Student %d:\n", i + 1);
        printf("Name: ");
        // Use fgets to safely read string input
        fgets(students[i].name, sizeof(students[i].name), stdin);
        // Remove the trailing newline character from fgets
        students[i].name[strcspn(students[i].name, "\n")] = '\0';
        printf("Roll Number: "); // Accept Roll Number
        scanf("%d", &students[i].roll_no);
        printf("Marks: ");   // Accept Marks
        scanf("%f", &students[i].marks);
        // Consume the leftover newline character from scanf
        // This is important before the next fgets call in the loop
        while (getchar() != '\n');
    }
    printf("\n--- Displaying Records of 5 Students ---\n");
    // Loop to display details of each student
    for (i = 0; i < 5; i++) {
        printf("\nDetails for Student %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;
}
```

47. What are nested structures? Write a C program to define a structure Employee that contains another structure Date (to store the joining date). Accept and display employee details.

Nested structures (or "structures within structures") occur when one structure is declared as a member inside another structure. This allows you to create more complex and logically organized data types by grouping related data that itself might be a composite type.

For example, an Employee structure might need to store a Date for their joining date. Instead of having separate day, month, and year members directly within Employee, it's more organized to define a Date structure first and then include a variable of type Date as a member of the Employee structure.

This approach improves code readability, modularity, and maintainability, as related data is kept together.

C Program: Employee Details with Nested Structure (Joining Date)

```c
#include <stdio.h>
#include <string.h> // For fgets and strcspn
// Define the inner structure: Date
struct Date {
    int day;
    int month;
    int year;
};
// Define the outer structure: Employee
// It contains members, including a variable of type 'struct Date'
struct Employee {
    char name[50];
    int employee_id;
    float salary;
    struct Date joining_date; // Nested structure member
};
int main() {
    struct Employee emp1; // Declare a variable of type struct Employee
```

```c
    printf("--- Enter Employee Details ---\n");
    printf("Enter Employee Name: ");
    fgets(emp1.name, sizeof(emp1.name), stdin);
    emp1.name[strcspn(emp1.name, "\n")] = '\0'; // Remove newline
    printf("Enter Employee ID: ");
    scanf("%d", &emp1.employee_id);
    printf("Enter Employee Salary: ");
    scanf("%f", &emp1.salary);
    // Consume the leftover newline character from scanf before reading date
    while (getchar() != '\n');
    printf("\nEnter Joining Date (DD MM YYYY):\n");
    printf("Day: ");
    scanf("%d", &emp1.joining_date.day); // Accessing nested member using dot operator
    printf("Month: ");
    scanf("%d", &emp1.joining_date.month);
    printf("Year: ");
    scanf("%d", &emp1.joining_date.year);
    printf("\n--- Displaying Employee Details ---\n");
    printf("Name: %s\n", emp1.name);
    printf("Employee ID: %d\n", emp1.employee_id);
    printf("Salary: %.2f\n", emp1.salary);
    printf("Joining Date: %02d/%02d/%d\n", // Use %02d for leading zeros for day/month
           emp1.joining_date.day,
           emp1.joining_date.month,
           emp1.joining_date.year);
    return 0;
}
```

48. Write a C program to read a file containing student records (Name, Roll No, Marks) and display students who scored more than 75 marks.

```c
#include <stdio.h> // For file operations (fopen, fclose, fscanf, printf)
#include <string.h> // For string functions (e.g., for dummy file creation)
// Define the Student structure
struct Student {
    char name[50];    // Student's name
    int roll_no;      // Student's roll number
    float marks;      // Student's marks
};
int main() {
    FILE *fptr; // File pointer
    char filename[] = "student_records.txt"; // Name of the file to read
    struct Student current_student; // Structure to hold one student's record at a time
    int students_displayed_count = 0; // Counter for students meeting the criteria
    // --- Create a dummy file for demonstration ---
    // In a real scenario, you would expect this file to already exist.
    // This part ensures the program is runnable out-of-the-box.
    fptr = fopen(filename, "w"); // Open in write mode to create/overwrite
    if (fptr == NULL) {
        printf("Error: Could not create dummy file '%s' for demonstration.\n", filename);
        return 1;
    }
    // Write sample student data to the file
    fprintf(fptr, "Alice 101 88.5\n");
    fprintf(fptr, "Bob 102 65.0\n");
    fprintf(fptr, "Charlie 103 90.0\n");
```

```c
    fprintf(fptr, "David 104 75.0\n"); // Marks are exactly 75, so not > 75
    fprintf(fptr, "Eve 105 70.5\n");
    fprintf(fptr, "Frank 106 80.0\n");
    fclose(fptr); // Close the dummy file
    printf("Sample student records created in '%s'.\n\n", filename);
    // --- Open the file for reading ---
    fptr = fopen(filename, "r"); // Open in read mode ("r")
    // Check if the file was opened successfully
    if (fptr == NULL) {
        printf("Error: Could not open file '%s' for reading.\n", filename);
        return 1; // Exit with an error code
    }
    printf("--- Students who scored more than 75 marks ---\n");
    printf("Name\tRoll No\tMarks\n");
    printf("----------------------------------------\n");
    // --- Read records and filter ---
    // fscanf reads formatted input from the file.
    // It returns the number of items successfully matched and assigned, or EOF
if end-of-file is reached.
    // We expect 3 items (string, int, float) per record.
    while (fscanf(fptr, "%s %d %f",
                  current_student.name,
                  &current_student.roll_no,
                  &current_student.marks) == 3) {
        // Check if the student's marks are greater than 75
        if (current_student.marks > 75.0) {
            // Display the details of students who meet the criteria
            printf("%s\t%d\t%.2f\n",
                   current_student.name,
                   current_student.roll_no,
```

```
                    current_student.marks);
            students_displayed_count++; // Increment counter for displayed
students
        }
    }
    // --- Close the file ---
    fclose(fptr); // Close the file to release resources
    // --- Final message ---
    if (students_displayed_count == 0) {
        printf("\nNo students found with marks greater than 75.\n");
    } else {
        printf("\nTotal %d student(s) displayed with marks greater than 75.\n",
students_displayed_count);
    }
    return 0; // Indicate successful program execution
}
```

49. Write a C program that takes user input (Name, Age, City) and writes it to a file in append mode. The program should then read the file and display its contents

```
#include <stdio.h>
#include <string.h> // For strcspn
int main() {
    FILE *fptr;
    char filename[] = "user_data.txt";
    char name[50];
    int age;
    char city[50];
    char buffer[200]; // Buffer to read lines when displaying

    // --- Part 1: Accept user input and write to file in append mode ---
```

```c
    printf("--- Enter Your Details ---\n");
    printf("Enter Name: ");
    fgets(name, sizeof(name), stdin);
    name[strcspn(name, "\n")] = '\0'; // Remove trailing newline
    printf("Enter Age: ");
    scanf("%d", &age);
    // Consume the leftover newline character from scanf
    while (getchar() != '\n');
    printf("Enter City: ");
    fgets(city, sizeof(city), stdin);
    city[strcspn(city, "\n")] = '\0'; // Remove trailing newline
    // Open file in append mode
    fptr = fopen(filename, "a");
    if (fptr == NULL) {
        printf("Error: Could not open/create file '%s' for writing.\n",
filename);
        return 1;
    }
    // Write data to the file
    fprintf(fptr, "Name: %s, Age: %d, City: %s\n", name, age, city);
    fclose(fptr);
    printf("Your details have been added to '%s'.\n\n", filename);
    // --- Part 2: Read the file and display its contents ---
    printf("--- Contents of '%s' ---\n", filename);
    // Open file in read mode
    fptr = fopen(filename, "r");
    if (fptr == NULL) {
        printf("Error: Could not open file '%s' for reading.\n", filename);
        return 1;
    }
```

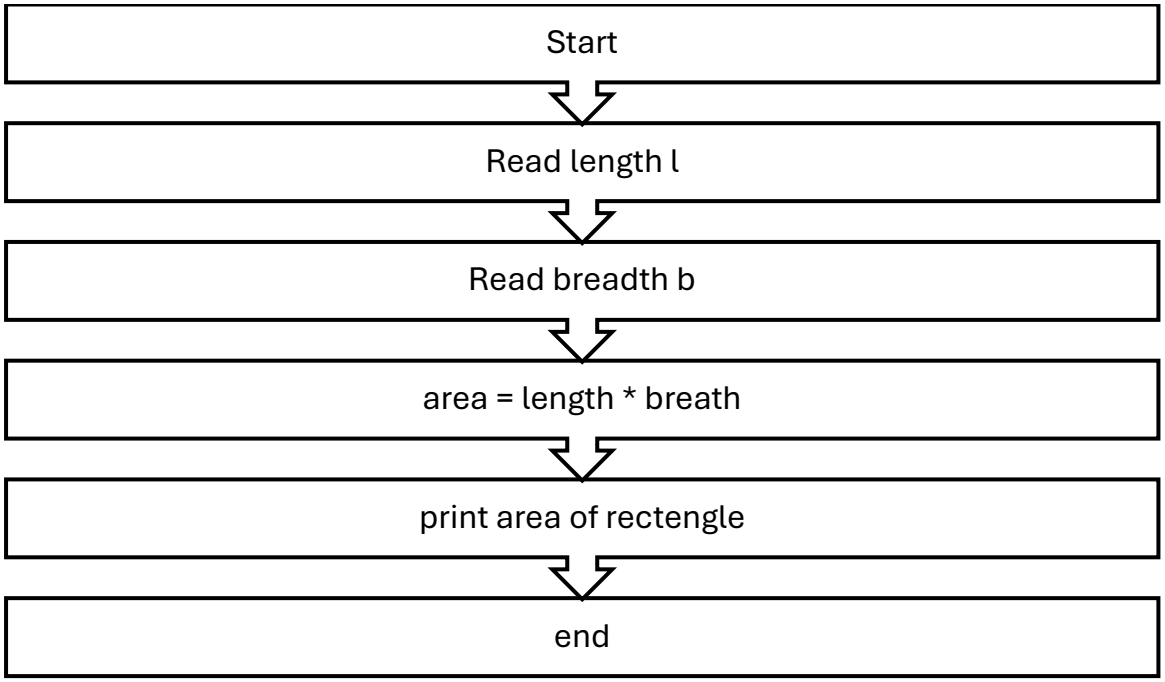
```c
    // Read and display each line from the file
    while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
        printf("%s", buffer);
    }
    fclose(fptr);
    printf("\n--- End of File Contents ---\n");
    return 0;
}
```

50. Define an operator in C. List and explain any four types of operators with examples.[co1] Common:

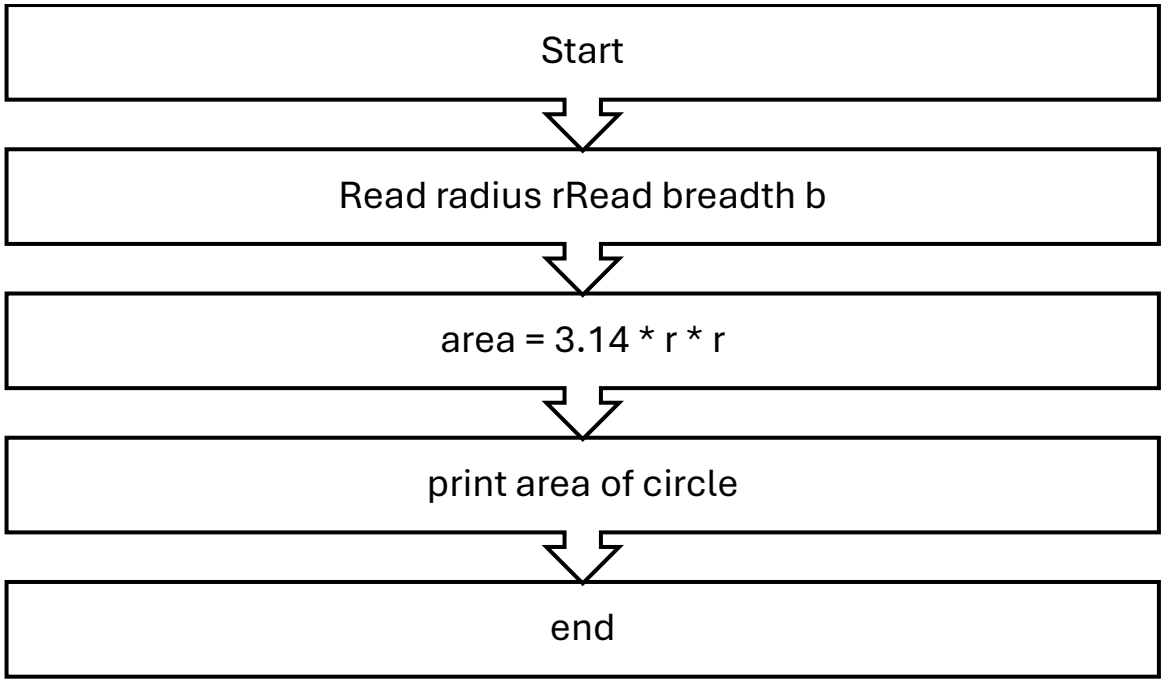
51. List and briefly explain the main components of a computer system.[co1] Common:

52. Draw and explain briefly two simple flowchart i) calculating the area of a rectangle and ii)Calculate the area of circle .[co1]

Area of Rectengle:



Area of Circle



53. Convert the following and explain the process:(a) Binary 10110.1100 to Decimal (b) Decimal 47 to Hexadecimal

54. Explain the difference between ‘for’, ‘while’ and ‘do-while’ loop in C with example programs.[co1]

55. What is variable? What are different types of variables which are called storage classes also?[co1][1+4]

What is a Variable?

In programming, a **variable** is a named storage location in a computer's memory that holds a value. It's essentially a placeholder for data that can change during the execution of a program. When you declare a variable, you tell the compiler to reserve a specific amount of memory for that variable, and you give that memory location a symbolic name.

Key characteristics of a variable:

- **Name:** A unique identifier to refer to the memory location.
- **Type:** Determines the kind of data it can store (e.g., integer, floating-point number, character, etc.) and the amount of memory it occupies.
- **Value:** The actual data stored in that memory location, which can be changed.
- **Address:** The specific memory location where the variable's value is stored.

Different Types of Variables (Storage Classes)

In C, the **storage class** of a variable determines its **scope**, **lifetime**, and **initial value**. It dictates where the variable will be stored in memory and how long it will persist. There are four main storage classes in C:

1. **auto (Automatic Storage Class)**
    - **Keyword:** auto (though rarely used explicitly as it's the default for local variables).
    - **Scope: Local** to the block or function in which it is declared. It's only accessible within that block.
    - **Lifetime:** Exists only while the block/function is executing. It's created when the block is entered and destroyed when the block is exited.
    - **Default Initial Value: Garbage value** (unpredictable). You must explicitly initialize auto variables if you want them to have a specific starting value.
    - **Memory Location:** Typically stored on the **stack**.
    - **Example:**

```
void myFunction() {
    int x; // 'x' is auto by default
    auto int y = 10; // Explicitly auto
}
```

2. **register (Register Storage Class)**
    - **Keyword:** register.
    - **Scope: Local** to the block or function in which it is declared.
    - **Lifetime:** Exists only while the block/function is executing.
    - **Default Initial Value: Garbage value**.
    - **Memory Location:** The compiler *attempts* to store register variables in **CPU registers** instead of main memory. This is a hint to the compiler, not a guarantee. If registers are not available, it defaults to auto storage.
    - **Purpose:** Used for variables that are accessed very frequently (like loop counters) to potentially speed up execution by avoiding memory access.
    - **Limitation:** You **cannot take the address** of a register variable using the & operator because it might not have a memory address.
    - **Example:**

```
void anotherFunction() {
    register int counter = 0; // Hint to store 'counter' in a register
    for (counter = 0; counter < 100; counter++) {
        // ...
    }
}
```

3. **static (Static Storage Class)**
    - **Keyword:** static.
    - **Scope:**
        - **Local static:** Local to the function/block where declared, but its lifetime is the entire program.
        - **Global static:** Local to the file where declared (file scope). Not accessible from other files.
    - **Lifetime:** Persists throughout the **entire duration of the program's execution**. It's initialized only once, at the start of the program, and retains its value between function calls.
    - **Default Initial Value: Zero** (for numeric types), NULL (for pointers), or empty string (for character arrays).
    - **Memory Location:** Stored in the **data segment** of the program's memory.
    - **Purpose:**
        - To create a local variable that retains its value across multiple function calls.
        - To restrict the visibility of a global variable or function to the file in which it is defined (internal linkage).
    - **Example:**

```
void countCalls() {
    static int callCount = 0; // Initialized once to 0, retains value
    callCount++;
    printf("Function called %d times.\n", callCount);
}

static int file_level_variable = 5; // Visible only within this file
```

4. **extern (External Storage Class)**
    - **Keyword:** extern.
    - **Scope: Global** (visible throughout the entire program across multiple files).
    - **Lifetime:** Persists throughout the **entire duration of the program's execution**.
    - **Default Initial Value: Zero** (if defined globally). When declared with extern, it doesn't allocate memory; it just declares that a variable is defined elsewhere.
    - **Memory Location:** Stored in the **data segment**.
    - **Purpose:** To declare a global variable or function that is defined in another source file. It tells the compiler that the variable exists, but its memory allocation and actual definition are handled in a different compilation unit.
    - **Example:**
        - **File1.c:**

```
int global_variable = 100; // Definition and initialization
```

        - **File2.c:**

```
extern int global_variable; // Declaration, refers to 'global_variable' in File1.c

void useGlobal() {

    printf("Global variable from another file: %d\n", global_variable);

}
```

56. Explain with an example how nested if statements work in C. Write a program to find the greatest of three numbers using nested if.[2+3]

Explanation of Nested if Statements in C

A **nested if statement** in C is an if (or if-else, else if) statement that is placed inside another if or else if or else block. This allows for checking multiple layers of conditions, where an inner condition is evaluated only if its outer condition is true.

Purpose: Nested if statements are used when a decision depends on another decision. They help you build more complex conditional logic, allowing your program to take different paths based on a hierarchy of conditions.

Syntax:

```
if (condition1) {
    // Code to execute if condition1 is true
    if (condition2) {
        // Code to execute if condition1 is true AND condition2 is true
    } else {
        // Code to execute if condition1 is true AND condition2 is false
    }
} else {
    // Code to execute if condition1 is false
}
```

In this structure, condition2 is only checked if condition1 evaluates to true.

C Program to Find the Greatest of Three Numbers using Nested if

This program will take three integer inputs from the user and use nested if statements to determine and display the largest among them.

```
#include <stdio.h> // Required for input/output operations
int main() {
    int num1, num2, num3; // Declare three integer variables to store the numbers
    printf("Enter three numbers: \n");
    printf("Enter first number: ");
    scanf("%d", &num1); // Read the first number
    printf("Enter second number: ");
    scanf("%d", &num2); // Read the second number
    printf("Enter third number: ");
    scanf("%d", &num3); // Read the third number
    // Outer if: Check if num1 is greater than or equal to num2
    if (num1 >= num2) {
        // If num1 is greater than or equal to num2, then compare num1 with num3
```

```
        if (num1 >= num3) {
            printf("%d is the greatest number.\n", num1);
        } else {
            // If num1 >= num2 is true, but num1 < num3, then num3 must be the
greatest
            printf("%d is the greatest number.\n", num3);
        }
    } else { // num1 < num2 (This block executes if num1 is NOT greater than or
equal to num2)
        // If num1 is less than num2, then compare num2 with num3
        if (num2 >= num3) {
            printf("%d is the greatest number.\n", num2);
        } else {
            // If num1 < num2 is true, and num2 < num3, then num3 must be the
greatest
            printf("%d is the greatest number.\n", num3);
        }
    }
    return 0; // Indicate successful program execution
}
```

How this program works with nested if:

1. **Outer if (num1 >= num2):**
    - If num1 is greater than or equal to num2, the program enters the first block. At this point, we know num1 is a strong candidate for the greatest.
    - If num1 is less than num2, the program enters the else block of the outer if. Now, num2 is a strong candidate for the greatest.
2. **Inner if statements:**
    - **Inside if (num1 >= num2) block:**
        - if (num1 >= num3): If num1 is also greater than or equal to num3, then num1 is confirmed as the greatest.

        - else: If num1 was num1 >= num2 but num1 < num3, it means num3 is the greatest (because num3 is greater than num1, and num1 is already known to be greater than or equal to num2).
    - **Inside else (where num1 < num2) block:**
        - if (num2 >= num3): If num2 is greater than or equal to num3, then num2 is confirmed as the greatest.
        - else: If num2 was num1 < num2 but num2 < num3, it means num3 is the greatest (because num3 is greater than num2, and num2 is already known to be greater than num1).

57. . Describe in detail how a C program is executed, from writing source code to generating the final output. Include the roles of editor, compiler, and linker.

The execution of a C program is a multi-stage process that transforms human-readable source code into machine-executable instructions and finally produces the desired output. This journey involves several critical tools: the editor, the compiler, and the linker.

Let's break down the process in detail:

1. Writing Source Code (Role of the Editor)

- **Action:** The programmer begins by writing the C program's logic in a text editor. This is the **source code**, which is a plain text file typically saved with a .c extension (e.g., myprogram.c).
- **Editor's Role:** The editor (like VS Code, Vim, Emacs, Notepad++, or even a simple Notepad) provides an environment for writing and managing the code. It facilitates text input, saving files, and often includes features like syntax highlighting (coloring keywords, variables, etc.), code formatting, and sometimes basic error checking.
- **Output:** A .c source file.

2. Compilation Process (Role of the Compiler)

The compiler is a complex software tool that translates the high-level C code into low-level machine-understandable instructions. This process is typically broken down into several phases:

a. Preprocessing (Pre-compiler)

- **Action:** The preprocessor is the first phase. It handles directives that begin with a # symbol.
- **Key Operations:**
    - **#include:** Inserts the content of specified header files (e.g., stdio.h for standard input/output functions) into the source code.

- **#define:** Replaces macros and symbolic constants with their defined values.
- **#ifdef, #ifndef, #endif:** Handles conditional compilation, including or excluding parts of the code based on defined symbols.

- **Compiler's Role:** It transforms the raw .c file into an expanded source file.
- **Output:** A .i file (intermediate file, rarely seen by the user) containing the expanded C code without any preprocessor directives.

b. Compilation (Compiler Proper)

- **Action:** This phase takes the preprocessed .i file and translates it into **assembly language**.
- **Key Operations:**
  - **Lexical Analysis:** Breaks the code into tokens (keywords, identifiers, operators, etc.).
  - **Syntax Analysis (Parsing):** Checks if the sequence of tokens forms a valid C program structure according to the language's grammar. It builds an abstract syntax tree (AST). If syntax errors are found, the compilation stops here.
  - **Semantic Analysis:** Checks for logical errors (e.g., type mismatches, undeclared variables).
  - **Intermediate Code Generation:** Converts the AST into a more abstract, machine-independent intermediate code.
  - **Code Optimization:** Attempts to improve the intermediate code for better performance (e.g., faster execution, smaller code size).
  - **Code Generation:** Translates the optimized intermediate code into target-specific assembly language instructions.
- **Compiler's Role:** Performs in-depth analysis and translates the high-level C logic into a form closer to the machine's instruction set.
- **Output:** An assembly file, typically with a .s or .asm extension (e.g., myprogram.s).

c. Assembly (Assembler)

- **Action:** The assembler takes the assembly language code generated by the compiler and translates it into **machine code** (binary instructions).
- **Key Operations:**
  - Converts assembly instructions (e.g., MOV EAX, 5) into their corresponding binary opcodes.
  - Resolves symbolic addresses to numerical memory addresses for local labels within the assembly file.

- **Compiler's Role (via Assembler):** Produces the raw machine instructions.
- **Output:** An **object file**, typically with a .o (Linux/macOS) or .obj (Windows) extension (e.g., myprogram.o). This file contains the machine code, data, a symbol table (listing functions and global variables defined/referenced in this file), and relocation information.

3. Linking Process (Role of the Linker)

- **Action:** The linker is the final crucial step that takes one or more object files (from your own code and potentially other modules) and combines them with code from standard libraries to produce a single, executable program.
- **Key Operations:**
    - **Symbol Resolution:** The most important task. Your code might call functions (like printf()) or use global variables that are defined in other object files or in the C standard library. The linker resolves these **external references** by finding the actual memory addresses of these functions and variables and patching your object code to point to them.
    - **Relocation:** Adjusts addresses in the object files to match their final locations in the executable's memory space.
    - **Library Inclusion:** Incorporates necessary routines from system libraries (e.g., libc for printf, scanf, etc.). This can be done either **statically** (copying the library code directly into the executable, making it larger but self-contained) or **dynamically** (linking to shared library files at runtime, making the executable smaller but dependent on those shared libraries).
- **Linker's Role:** To gather all necessary machine code components and glue them together into a complete, runnable program.
- **Output:** An **executable file** (e.g., a.out by default on Linux/macOS, myprogram.exe on Windows).

4. Loading and Execution (Role of the Operating System)

- **Action:** Once the executable file is generated, the operating system (OS) takes over to run it.
- **Loader's Role (part of OS):**
    - Loads the executable file from disk into the computer's Random Access Memory (RAM).
    - Allocates memory for the program's segments (code, data, stack, heap).
    - Performs any final runtime relocations (especially for dynamically linked libraries).
    - Sets up the program's initial execution environment (e.g., initializing CPU registers, setting up the stack pointer).
    - Transfers control to the program's entry point, which is typically the main() function in C.

- **Execution:** The CPU then starts executing the machine instructions from the main() function. The program interacts with the OS for tasks like input/output, memory management, etc.
- **Output:** The program interacts with standard output (like the console) to display results, or it might write to files, or perform other operations that generate an observable outcome.

58. State the difference between entry-controlled and exit-controlled loop.[4] State the difference between count controlled and condition controlled loop.[1]

Feature	Entry-Controlled Loop	Exit-Controlled Loop
Condition Check	Condition is checked before the loop body is executed.	Condition is checked after the loop body is executed.
Execution Guarantee	The loop body may not execute even once if the condition is initially false.	The loop body is guaranteed to execute at least once, regardless of the initial condition.
Syntax Examples in C	for loop, while loop	do-while loop
Flow of Control	Condition -> Body -> Condition -> Body ...	Body -> Condition -> Body -> Condition ...
Typical Use Cases	When you need to iterate a specific number of times (for), or when you might not need to execute the loop at all (while).	When you need to execute the loop body at least once (e.g., getting valid user input).

Example in C:

Entry-Controlled (while loop):

```c
int i = 5;
while (i < 5) { // Condition is false initially
    printf("This will NOT be printed.\n");
    i++;
}
printf("Loop finished.\n"); // Output: Loop finished.
```

Here, i < 5 is false from the start, so the printf inside the loop never runs.

Exit-Controlled (do-while loop):

```
int i = 5;
do {
    printf("This will be printed at least once. i = %d\n", i);
    i++;
} while (i < 5); // Condition is checked AFTER the body
printf("Loop finished.\n");
// Output: This will be printed at least once. i = 5\nLoop finished.
```

Here, even though i < 5 is false after the first iteration, the loop body executes once because the condition is checked at the end.

Difference Between Count-Controlled and Condition-Controlled Loops [1 mark]

- **Count-Controlled Loop:** A loop that executes a **fixed, predetermined number of times**. The number of iterations is known before the loop starts.
    - **Example:** A for loop iterating from 1 to 10. for (int i = 1; i <= 10; i++)
- **Condition-Controlled Loop:** A loop that executes an **unspecified number of times**, continuing as long as a certain condition remains true. The number of iterations is not known in advance and depends on runtime factors.
    - **Example:** A while loop that continues until the user enters 'quit'. while (user_input != 'quit')

59. What is two-way selection statement? Explain if, if-else, and cascaded if-else with examples.[5]

What is a Two-Way Selection Statement?

A **selection statement** (or conditional statement) in C allows your program to make decisions and execute different blocks of code based on whether a specified condition is true or false.

A **two-way selection statement** implies that there are primarily two possible paths or outcomes for a decision:

1. A certain block of code executes if a condition is met.
2. An alternative block of code executes if the condition is *not* met.

While the if statement by itself can be considered a single-way selection (execute if true, do nothing if false), the if-else construct truly embodies a two-way selection by providing distinct paths for both true and false outcomes. The "cascaded if-else" or if-else if-else ladder extends this concept to multiple potential paths.

Types of if Statements in C

1. The if Statement (Single-Way Selection)

- **Purpose:** The simplest form of a conditional statement. It executes a block of code *only if* a specified condition evaluates to true. If the condition is false, the block is skipped, and the program continues with the statement immediately following the if block.
- **Syntax:**

```
if (condition) {
    // Code to execute if condition is true
}
// Program continues here
```

- **Example:** Check if a number is positive.

```
#include <stdio.h>
int main() {
    int num = 10;
    if (num > 0) {
        printf("The number %d is positive.\n", num);
    }
    printf("End of if statement example.\n");
    return 0;
}
```

Output:

```
The number 10 is positive.
End of if statement example.
```

2. The if-else Statement (True Two-Way Selection)

- **Purpose:** This statement provides two distinct execution paths. One block of code is executed if the condition is true, and a *different* block of code (following else) is executed if the condition is false. The two blocks are mutually exclusive.

- **Syntax:**

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
// Program continues here
```

- **Example:** Check if a number is even or odd.

```
#include <stdio.h>
int main() {
    int num = 7;
    if (num % 2 == 0) { // Check if remainder when divided by 2 is 0
        printf("%d is an even number.\n", num);
    } else {
        printf("%d is an odd number.\n", num);
    }
    printf("End of if-else statement example.\n");
    return 0;
}
```

Output:

7 is an odd number.

End of if-else statement example.

3. Cascaded if-else (or if-else if-else Ladder / Multi-way Selection)

- **Purpose:** This construct is used when there are multiple conditions to check sequentially, and each condition leads to a different course of action. The conditions are evaluated from top to bottom. As soon as a condition evaluates to true, its corresponding block of code is executed, and the rest of the ladder is skipped. If none of the if or else if conditions are true, the final else block (if present) is executed as a default.

- **Syntax:**

```
if (condition1) {
    // Code if condition1 is true
} else if (condition2) {
    // Code if condition1 is false AND condition2 is true
} else if (condition3) {
    // Code if condition1 and condition2 are false AND condition3 is true
} else {
    // Code if all preceding conditions are false
}
// Program continues here
```

- **Example:** Assign a grade based on marks.

```
#include <stdio.h>
int main() {
    int marks = 85;
    char grade;
    if (marks >= 90) {
        grade = 'A';
    } else if (marks >= 80) { // Only checked if marks < 90
        grade = 'B';
    } else if (marks >= 70) { // Only checked if marks < 80
        grade = 'C';
    } else if (marks >= 60) { // Only checked if marks < 70
        grade = 'D';
    } else { // If none of the above conditions are met
        grade = 'F';
    }
    printf("Marks: %d, Grade: %c\n", marks, grade);
    printf("End of cascaded if-else example.\n");
```

```
    return 0;
}
```

Output:

```
Marks: 85, Grade: B
End of cascaded if-else example.
```

If marks were 55, the output would be Marks: 55, Grade: F.

60. Explain the difference between = and == operators with examples.

In C programming, = and == are two distinct operators with fundamentally different purposes. Understanding their difference is crucial to avoid common programming errors.

1. = (Assignment Operator)

- **Purpose:** The single equals sign (=) is the **assignment operator**. Its primary function is to **assign** the value of the operand on the right-hand side (RHS) to the variable on the left-hand side (LHS).
- **Behavior:** It takes the value from the RHS expression and *stores* it into the memory location designated by the LHS variable. The LHS must be a modifiable lvalue (something that can hold a value, like a variable).
- **Return Value:** An assignment expression itself evaluates to the value that was assigned.
- **Analogy:** Think of it like putting something *into* a box.

Example:

```
#include <stdio.h>
int main() {
    int score = 0; // Declare an integer variable 'score' and assign it the value 0
    score = 100;   // Assign the value 100 to the variable 'score'
    printf("After score = 100; score is: %d\n", score); // Output: score is: 100
    int x = 5;
    int y = 10;
    x = y;         // Assign the value of 'y' (which is 10) to 'x'
    printf("After x = y; x is: %d, y is: %d\n", x, y); // Output: x is: 10, y is: 10
```

```
    return 0;
}
```

2. == (Equality Operator)

- **Purpose:** The double equals sign (==) is the **equality operator** (also known as the comparison operator). Its purpose is to **compare** the values of the two operands (one on the LHS and one on the RHS) to check if they are equal.
- **Behavior:** It evaluates to a boolean-like result:
    - 1 (or any non-zero value, typically used to represent true) if the values are equal.
    - 0 (to represent false) if the values are not equal.
- **Return Value:** An integer (1 for true, 0 for false).
- **Analogy:** Think of it like asking a question: "Is this box holding the same thing as that box?"

Example:

```
#include <stdio.h>
int main() {
    int a = 10;
    int b = 20;
    int c = 10;
    // Compare 'a' and 'b'
    if (a == b) { // Is 10 equal to 20? No (false)
        printf("a is equal to b\n");
    } else {
        printf("a is NOT equal to b\n"); // This will be printed
    }
    // Compare 'a' and 'c'
    if (a == c) { // Is 10 equal to 10? Yes (true)
        printf("a is equal to c\n"); // This will be printed
    } else {
        printf("a is NOT equal to c\n");
    }
```

```
    // Direct output of comparison result
    printf("Result of (5 == 5): %d\n", (5 == 5));   // Output: 1 (true)
    printf("Result of (10 == 3): %d\n", (10 == 3)); // Output: 0 (false)
    return 0;
}
```

Key Difference and Common Mistake

The most important distinction is:

- = **assigns** a value.
- == **compares** values.

A very common and subtle bug in C programming (especially for beginners) is accidentally using = instead of == inside a conditional statement like an if loop.

Common Mistake Example:

```
#include <stdio.h>
int main() {
    int myValue = 5;
    // DANGEROUS MISTAKE: Using '=' instead of '=='
    if (myValue = 0) { // This assigns 0 to myValue, and the expression (myValue=0) evaluates to 0 (false)
        printf("Condition is true (myValue is 0)\n");
    } else {
        printf("Condition is false (myValue is not 0)\n"); // This will be printed
    }
    printf("myValue after the 'if' statement: %d\n", myValue); // myValue is now 0
    myValue = 5; // Reset for next example
    // Correct way to compare
    if (myValue == 0) { // This compares myValue with 0. It's false.
        printf("Condition is true (myValue is 0)\n");
    } else {
```

```
        printf("Condition is false (myValue is not 0)\n"); // This will be
printed
    }
    printf("myValue after the correct 'if' statement: %d\n", myValue);
    // myValue is still 5
    return 0;
}
```

In the first if statement, myValue = 0 is an assignment. The value assigned (0) is then treated as the condition. Since 0 is considered false in C's boolean context, the else block executes. Crucially, myValue itself has been changed to 0. This can lead to very hard-to-find bugs!

61. Explain how a switch statement works with an example of menu-based program

The switch statement in C is a powerful control flow statement that allows you to select one of many code blocks to be executed. It's often used as an alternative to a long if-else if-else ladder when you are testing a single expression against multiple possible constant values.

How a switch Statement Works

The switch statement evaluates an **expression** (which must result in an integer value, e.g., int, char, enum). The value of this expression is then compared against the values of various **case labels**.

Here's a breakdown of its components and behavior:

1. **switch (expression):**
    - The expression inside the parentheses is evaluated. Its result must be an integer type (like int, char, short, long, or enum). Floating-point numbers are not allowed.
2. **case constant_value::**
    - Each case label is followed by a constant_value (which must be a unique, literal constant, not a variable or expression).
    - If the value of the switch expression matches a constant_value of a case label, the control of the program jumps directly to the code block associated with that case.
3. **break;:**
    - The break statement is crucial inside a switch block. When encountered, it terminates the switch statement and transfers control to the statement immediately following the switch block.

- **Without break (Fall-through):** If break is omitted, execution will "fall through" to the next case label's code block, and continue executing statements sequentially until a break is found or the end of the switch block is reached. This is sometimes intentional but more often a common source of bugs.

4. **default: (Optional):**
    - The default label is optional. If no case constant matches the value of the switch expression, the control of the program jumps to the default block (if present).
    - There can be only one default label in a switch statement.
    - It doesn't necessarily need a break if it's the last block, but it's good practice to include one for consistency and if new case labels are added later.

Flow of Control:

1. The switch expression is evaluated once.
2. The result is compared with case constant values.
3. If a match is found, execution jumps to that case label.
4. Execution continues from that case label downwards.
5. If a break statement is encountered, the switch statement terminates.
6. If no case matches and a default label is present, execution jumps to the default block.
7. If no case matches and no default is present, nothing inside the switch block is executed, and control passes to the statement after the switch.

Example: Menu-Based Program using switch

This program will present a simple menu of arithmetic operations to the user and perform the chosen operation using a switch statement.

```
#include <stdio.h> // For input/output functions
int main() {
    int choice;
    double num1, num2, result;
    printf("--- Simple Calculator Menu ---\n");
    printf("1. Addition\n");
    printf("2. Subtraction\n");
    printf("3. Multiplication\n");
    printf("4. Division\n");
```

```
    printf("5. Exit\n");
    printf("------------------------------\n");
    printf("Enter your choice (1-5): ");
    scanf("%d", &choice); // Read user's choice
    // Check if the choice requires inputting two numbers
    if (choice >= 1 && choice <= 4) {
        printf("Enter first number: ");
        scanf("%lf", &num1);
        printf("Enter second number: ");
        scanf("%lf", &num2);
    }
    // Use switch statement to perform action based on user's choice
    switch (choice) {
        case 1: // If choice is 1
            result = num1 + num2;
            printf("Result: %.2lf + %.2lf = %.2lf\n", num1, num2, result);
            break; // Exit the switch statement
        case 2: // If choice is 2
            result = num1 - num2;
            printf("Result: %.2lf - %.2lf = %.2lf\n", num1, num2, result);
            break; // Exit the switch statement
        case 3: // If choice is 3
            result = num1 * num2;
            printf("Result: %.2lf * %.2lf = %.2lf\n", num1, num2, result);
            break; // Exit the switch statement
        case 4: // If choice is 4
            if (num2 != 0) { // Check for division by zero
                result = num1 / num2;
                printf("Result: %.2lf / %.2lf = %.2lf\n", num1, num2, result);
```

```
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break; // Exit the switch statement
        case 5: // If choice is 5
            printf("Exiting the calculator. Goodbye!\n");
            break; // Exit the switch statement
        default: // If choice does not match any of the above cases
            printf("Invalid choice. Please enter a number between 1 and 5.\n");
            break; // Good practice, though not strictly necessary as it's the
last case
    }
    return 0; // Indicate successful program execution
}
```

How the switch statement works in this example:

1. The program displays a menu and prompts the user for a choice (an integer).
2. The value of choice is passed to the switch statement.
3. The switch statement then compares choice with the constant_value of each case label (1, 2, 3, 4, 5).
4. **If choice is 1:** The code under case 1: executes, performing addition. The break; statement then terminates the switch block, and the program proceeds to return 0;.
5. **If choice is 4:** The code under case 4: executes, performing division (with a zero-check). The break; terminates the switch.
6. **If choice is 5:** The exit message is printed, and the break; terminates.
7. **If choice is any other number** (e.g., 0, 6, -1): None of the case labels match, so the default: block executes, printing an "Invalid choice" message. The break; then terminates the switch.

62. Write a C program that accepts a number from the user and checks whether it is: • Even or Odd • Positive or Negative • Divisible by 5 or not Use appropriate conditional statements.[CO1][2+2+1]

```c
#include <stdio.h> // Include standard input/output library
int main() {
    int number; // Declare an integer variable to store the user's input
    // Prompt the user to enter a number
    printf("Enter an integer: ");
    scanf("%d", &number); // Read the integer from the user
    // --- Check 1: Even or Odd ---
    printf("\n--- Even or Odd Check ---\n");
    if (number % 2 == 0) {
        printf("%d is an Even number.\n", number);
    } else {
        printf("%d is an Odd number.\n", number);
    }
    // --- Check 2: Positive, Negative, or Zero ---
    printf("\n--- Positive, Negative, or Zero Check ---\n");
    if (number > 0) {
        printf("%d is a Positive number.\n", number);
    } else if (number < 0) {
        printf("%d is a Negative number.\n", number);
    } else { // If it's not > 0 and not < 0, it must be 0
        printf("%d is Zero.\n", number);
    }
    // --- Check 3: Divisible by 5 or not ---
    printf("\n--- Divisibility by 5 Check ---\n");
    if (number % 5 == 0) {
        printf("%d is Divisible by 5.\n", number);
```

```
    } else {
        printf("%d is NOT Divisible by 5.\n", number);
    }
    return 0; // Indicate successful program execution
}
```

63. A. Create a program using nested loops that generates the following pattern (example for n = 5) [5]

```
*
**
***
****
*****
```

```
#include <stdio.h>
int main() {
    int n = 5; // Number of rows for the pattern (as per example)
    int i, j;  // Loop counters
    printf("--- Star Pattern (n = %d) ---\n", n);
    // Outer loop: Controls the number of rows
    // It runs from 1 to n
    for (i = 1; i <= n; i++) {
        // Inner loop: Controls the number of stars in the current row
        // It runs from 1 to the current row number (i)
        for (j = 1; j <= i; j++) {
            printf("*"); // Print a star
        }
        printf("\n"); // Move to the next line after printing all stars for the current row
    }
    return 0;
}
```

B. Create a pattern shown below using loops[5]

1

2 3

4 5 6

7 8 9 10

11 2 13 14 15

```c
#include <stdio.h>
int main() {
    int n = 5;        // Number of rows for the pattern (as per example)
    int i, j;         // Loop counters
    int current_number = 1;
    printf("--- Number Pattern (n = %d) ---\n", n);
    // Outer loop: Controls the number of rows
    // It runs from 1 to n
    for (i = 1; i <= n; i++) {
        // Inner loop: Controls how many numbers are printed in the current row
        // It runs from 1 to the current row number (i)
        for (j = 1; j <= i; j++) {
            printf("%d ", current_number); // Print the sequential number
            current_number++;   // Increment the number for the next print
        }
       printf("\n");
    }
    return 0;
}
```

64. Create a calculator(+, -, *, / , %) using switch case(create it in menu driven format). Example: press 1 for addition Press 2 for subtraction Press 3 for multiplication Press 4 for division Press 5 for modulus Press 6 for menu Press 7 for exit and also provide all the options in output.

```c
#include <stdio.h>

// Function to display the calculator menu
void displayMenu() {
    printf("\n--- Calculator Menu ---\n");
    printf("1. Addition (+)\n");
    printf("2. Subtraction (-)\n");
    printf("3. Multiplication (*)\n");
    printf("4. Division (/)\n");
    printf("5. Modulus (%%)\n"); // Use %% to print a literal % sign
    printf("6. Show Menu Again\n");
    printf("7. Exit\n");
    printf("-----------------------\n");
}
int main() {
    int choice;
    double num1, num2; // Use double for general arithmetic operations
    int int_num1, int_num2; // For modulus operator, which requires integers
    int exit_program = 0; // Flag to control program loop
    displayMenu(); // Display the menu for the first time
    // Loop indefinitely until the user chooses to exit
    do {
        printf("Enter your choice (1-7): ");
        // Ensure that scanf successfully reads an integer.
        // If not, clear the input buffer to prevent infinite loops.
        if (scanf("%d", &choice) != 1) {
```

```
            printf("Invalid input. Please enter a number.\n");
      // Clear input buffer: read and discard characters until a newline or EOF
            while (getchar() != '\n');
            continue; // Skip to the next iteration of the loop
        }
       // Handle cases that don't require two numbers first (Show Menu,Exit,
Invalid)
        if (choice == 6) {
            displayMenu();
            continue; // Go back to the start of the loop to get choice
        } else if (choice == 7) {
            exit_program = 1; // Set flag to exit loop
            printf("Exiting the calculator. Goodbye!\n");
            break; // Exit the do-while loop
        } else if (choice < 1 || choice > 7) {
            printf("Invalid choice. Please enter a number between 1 and 7.\n");
            continue; // Go back to the start of the loop
        }
        // For operations 1-5, get two numbers
        printf("Enter first number: ");
        if (scanf("%lf", &num1) != 1) {
            printf("Invalid input. Please enter a valid number.\n");
            while (getchar() != '\n');
            continue;
        }
        printf("Enter second number: ");
        if (scanf("%lf", &num2) != 1) {
            printf("Invalid input. Please enter a valid number.\n");
            while (getchar() != '\n');
            continue;
```

```
        }
        // Use switch statement to perform the chosen operation
        switch (choice) {
           case 1: // Addition
            printf("Result: %.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);
                break;
           case 2: // Subtraction
            printf("Result: %.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);
                break;
           case 3: // Multiplication
            printf("Result: %.2lf * %.2lf = %.2lf\n", num1, num2, num1 * num2);
                break;
            case 4: // Division
                if (num2 != 0) { // Check for division by zero
                printf("Result: %.2lf / %.2lf = %.2lf\n", num1, num2, num1 /
num2);
                } else {
                    printf("Error: Division by zero is not allowed.\n");
                }
                break;
            case 5: // Modulus
                // Modulus operator (%) works only with integer operands in C
                int_num1 = (int)num1;
                int_num2 = (int)num2;
                printf("Note: Modulus operator (%%) works only on integers.
Using integer parts of %.2lf and %.2lf.\n", num1, num2);
                if (int_num2 != 0) {
                    printf("Result: %d %% %d = %d\n", int_num1, int_num2,
int_num1 % int_num2);
                } else {
```

```
                    printf("Error: Modulus by zero is not allowed.\n");
                }
                break;
            }
            printf("\n");
        } while (exit_program == 0); // Continue loop until exit_program is 1
        return 0;
}
```

65. Explain the concept of operator precedence and associativity in C with table. Why is it important in expression evaluation? Provide examples.

66. Write a program in string to show string length, comparison of two string, copy to another string up to 4 characters, string lower and string reverse.

```
#include <stdio.h>
#include <string.h> // strlen, strcmp, strncpy, strcpy, strcspn
#include <ctype.h>  // tolower
// Removes trailing newline from fgets input
void removeNewline(char *str) {
    str[strcspn(str, "\n")] = '\0';
}
// Converts string to lowercase
void stringToLower(char *str) {
    for (int i = 0; str[i] != '\0'; i++) {
        str[i] = tolower(str[i]);
    }
}
// Reverses a string
void stringReverse(char *str) {
    int length = strlen(str);
    char temp;
```

```
    for (int i = 0, j = length - 1; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}
int main() {
    char str1[100], str2[100], copy_str[100], lower_str[100], reverse_str[100];
    int len, cmp_result;
    printf("--- String Operations Demonstration ---\n\n");
    // --- 1. String Length ---
    printf("Enter a string to find its length: ");
    fgets(str1, sizeof(str1), stdin);
    removeNewline(str1);
    len = strlen(str1);
    printf("Length of \"%s\" is: %d\n\n", str1, len);
    // --- 2. Comparison of Two Strings ---
    printf("Enter first string for comparison: ");
    fgets(str1, sizeof(str1), stdin);
    removeNewline(str1);
    printf("Enter second string for comparison: ");
    fgets(str2, sizeof(str2), stdin);
    removeNewline(str2);
    cmp_result = strcmp(str1, str2);
    if (cmp_result == 0) {
        printf("\"%s\" and \"%s\" are equal.\n\n", str1, str2);
    } else if (cmp_result < 0) {
        printf("\"%s\" is lexicographically smaller than \"%s\".\n\n", str1,
str2);
    } else {
```

```
        printf("\"%s\" is lexicographically greater than \"%s\".\n\n", str1,
str2);
    }
    // --- 3. Copy to another string up to 4 characters ---
    printf("Original string for partial copy: \"%s\"\n", str1);
    strncpy(copy_str, str1, 4); // Copy up to 4 chars
    copy_str[4] = '\0'; // Manual null-termination is crucial for strncpy
    printf("Copied (up to 4 chars) string: \"%s\"\n\n", copy_str);
    // --- 4. String Lowercase ---
    printf("Original string for lowercase conversion: \"%s\"\n", str1);
    strcpy(lower_str, str1); // Copy for modification
    stringToLower(lower_str);
    printf("Lowercase string: \"%s\"\n\n", lower_str);
    // --- 5. String Reverse ---
    printf("Original string for reversal: \"%s\"\n", str1);
    strcpy(reverse_str, str1); // Copy for modification
    stringReverse(reverse_str);
    printf("Reversed string: \"%s\"\n\n", reverse_str);
    return 0;
}
```

67. Explain the concept of a nested loop in matrix operations with an example. Write the program of transpose of a matrix with suitable output.

```
#include <stdio.h>
int main() {
    int rows, cols; // Dimensions of the original matrix
    int matrix[10][10]; // Original matrix (max 10x10)
    int transpose[10][10]; // Transposed matrix (max 10x10)
    int i, j; // Loop counters
    // --- 1. Accept matrix dimensions ---
```

```c
    printf("Enter the number of rows (max 10): ");
    scanf("%d", &rows);
    printf("Enter the number of columns (max 10): ");
    scanf("%d", &cols);
    // Input validation for dimensions
    if (rows <= 0 || rows > 10 || cols <= 0 || cols > 10) {
        printf("Error: Rows and columns must be between 1 and 10.\n");
        return 1;
    }
    // --- 2. Accept elements of the original matrix ---
    printf("\nEnter elements of the matrix:\n");
    // Outer loop for rows
    for (i = 0; i < rows; i++) {
        // Inner loop for columns
        for (j = 0; j < cols; j++) {
            printf("Enter element matrix[%d][%d]: ", i, j);
            scanf("%d", &matrix[i][j]);
        }
    }
    // --- 3. Display the Original Matrix ---
    printf("\nOriginal Matrix (%dx%d):\n", rows, cols);
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            printf("%d\t", matrix[i][j]); // Print element followed by a tab
        }
        printf("\n"); // Newline after each row
    }
    // --- 4. Calculate the Transpose ---
    // The transpose will have 'cols' rows and 'rows' columns
```

```
    // Outer loop for rows of the TRANSPOSE matrix (which are columns of
original)
    for (i = 0; i < cols; i++) {
        // Inner loop for columns of the TRANSPOSE matrix (which are rows of
original)
        for (j = 0; j < rows; j++) {
            transpose[i][j] = matrix[j][i]; // Key step: swap row and column
indices
        }
    }
    // --- 5. Display the Transposed Matrix ---
    printf("\nTransposed Matrix (%dx%d):\n", cols, rows);
    for (i = 0; i < cols; i++) { // Iterate through rows of transpose
        for (j = 0; j < rows; j++) { // Iterate through columns of transpose
            printf("%d\t", transpose[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

68. Write the bubble sort algorithm. Compute the bubble sort: 90, 50, 70, 10, 60, 30.

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

How it works:

1. Comparison and Swap: It starts at the beginning of the list and compares the first two elements. If the first is greater than the second, they are swapped.
2. Move to Next Pair: It then moves to the next pair of elements (second and third) and repeats the comparison and swap.

3. Iterate through List: This process continues until the end of the list is reached. After the first pass, the largest element will "bubble up" to its correct position at the end of the unsorted portion of the list.
4. Repeat Passes: The algorithm then repeats the entire process for the remaining unsorted portion of the list, reducing the size of the unsorted portion by one element in each pass.
5. Termination: The sorting stops when a pass completes without any swaps, indicating that the array is fully sorted.

Key Characteristics:

- Simple: Easy to understand and implement.
- Inefficient: Not suitable for large datasets as its average and worst-case time complexity is O(n2), where n is the number of items.
- Stable: It preserves the relative order of equal elements.

Computation of Bubble Sort: 90,50,70,10,60,30

Let's sort the array: [90, 50, 70, 10, 60, 30]

Initial Array: [90, 50, 70, 10, 60, 30]

Pass 1: (Largest element bubbles to the end)

- Compare 90 and 50: 90>50, swap.

[50, 90, 70, 10, 60, 30]

- Compare 90 and 70: 90>70, swap.

[50, 70, 90, 10, 60, 30]

- Compare 90 and 10: 90>10, swap.

[50, 70, 10, 90, 60, 30]

- Compare 90 and 60: 90>60, swap.

[50, 70, 10, 60, 90, 30]

- Compare 90 and 30: 90>30, swap.

[50, 70, 10, 60, 30, 90]

End of Pass 1. 90 is now in its correct position.

Pass 2: (Second largest element bubbles up)

- Compare 50 and 70: 50<70, no swap.

[50, 70, 10, 60, 30, 90]

- Compare 70 and 10: 70>10, swap.

[50, 10, 70, 60, 30, 90]

- Compare 70 and 60: 70>60, swap.

[50, 10, 60, 70, 30, 90]

- Compare 70 and 30: 70>30, swap.

[50, 10, 60, 30, 70, 90]

End of Pass 2. 70 is now in its correct position.

Pass 3: (Third largest element bubbles up)

- Compare 50 and 10: 50>10, swap.

[10, 50, 60, 30, 70, 90]

- Compare 50 and 60: 50<60, no swap.

[10, 50, 60, 30, 70, 90]

- Compare 60 and 30: 60>30, swap.

[10, 50, 30, 60, 70, 90]

End of Pass 3. 60 is now in its correct position.

Pass 4: (Fourth largest element bubbles up)

- Compare 10 and 50: 10<50, no swap.

[10, 50, 30, 60, 70, 90]

- Compare 50 and 30: 50>30, swap.

[10, 30, 50, 60, 70, 90]

End of Pass 4. 50 is now in its correct position.

Pass 5: (Fifth largest element bubbles up)

- Compare 10 and 30: 10<30, no swap.

[10, 30, 50, 60, 70, 90]

End of Pass 5. 30 is now in its correct position.

Pass 6: (Check for sorted array - no swaps needed)

- No swaps occur in this pass. The array is sorted. Final Sorted Array: [10, 30, 50, 60, 70, 90]

69. Write a program to sort an array using selection sort with output. Explain each step using algorithm and an example.

Selection Sort Algorithm

Selection Sort is an in-place comparison sorting algorithm. It works by repeatedly finding the minimum element (or maximum, depending on the sorting order) from the unsorted part of the list and putting it at the beginning of the unsorted part.

How it works:

1. Find Minimum: Start by assuming the first element in the unsorted portion is the minimum. Iterate through the rest of the unsorted portion to find the actual minimum element.
2. Swap: Once the minimum element is found, swap it with the element at the current position (the beginning of the unsorted portion).
3. Shrink Unsorted Portion: The element that was just placed at the current position is now considered sorted. The unsorted portion of the list shrinks by one element from the left.
4. Repeat: Repeat steps 1-3 for the remaining unsorted portion until the entire list is sorted.

Key Characteristics:

- In-place: It sorts the array by modifying the input array directly, without requiring extra space.
- Unstable: It does not preserve the relative order of elements with equal values.
- Inefficient: Like Bubble Sort, its time complexity is O(n2) in all cases (best, average, and worst), making it unsuitable for large datasets. This is because it always performs N passes, and in each pass, it scans the unsorted part of the array.

Step-by-Step Example: Sorting 90,50,70,10,60,30

Let's sort the array: [90, 50, 70, 10, 60, 30]

Initial Array: [90, 50, 70, 10, 60, 30]

Pass 1: (Find the minimum in [90, 50, 70, 10, 60, 30] and place it at index 0)

- Current unsorted portion: [90, 50, 70, 10, 60, 30]
- Minimum element found: 10 (at index 3)
- Swap 90 (at index 0) with 10 (at index 3).
- Array after Pass 1: [10, 50, 70, 90, 60, 30]

Pass 2: (Find the minimum in [50, 70, 90, 60, 30] and place it at index 1)

- Current unsorted portion: [50, 70, 90, 60, 30]
- Minimum element found: 30 (at index 5 in original array, which is index 4 in the current unsorted portion)
- Swap 50 (at index 1) with 30 (at index 4).
- Array after Pass 2: [10, 30, 70, 90, 60, 50]

Pass 3: (Find the minimum in [70, 90, 60, 50] and place it at index 2)

- Current unsorted portion: [70, 90, 60, 50]
- Minimum element found: 50 (at index 5)
- Swap 70 (at index 2) with 50 (at index 5).
- Array after Pass 3: [10, 30, 50, 90, 60, 70]

Pass 4: (Find the minimum in [90, 60, 70] and place it at index 3)

- Current unsorted portion: [90, 60, 70]
- Minimum element found: 60 (at index 4)
- Swap 90 (at index 3) with 60 (at index 4).
- Array after Pass 4: [10, 30, 50, 60, 90, 70]

Pass 5: (Find the minimum in [90, 70] and place it at index 4)

- Current unsorted portion: [90, 70]
- Minimum element found: 70 (at index 5)
- Swap 90 (at index 4) with 70 (at index 5).
- Array after Pass 5: [10, 30, 50, 60, 70, 90]

Final Sorted Array: [10, 30, 50, 60, 70, 90]

70. Write a program to search an element in an array using linear search. Explain the difference between linear search and binary search with an example.

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int target) {

    // Loop through each element of the array from index 0 to n-1

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

        // Check if the current element is equal to the target
```

```c
        if (arr[i] == target) {
            // If a match is found, return the current index
            return i;
        }
    }
    // If the loop completes without finding the target,
    // it means the element is not present in the array.
    return -1;
}
int main() {
    // Define an example array
    int my_array[] = {90, 50, 70, 10, 60, 30};
    // Calculate the number of elements in the array
    // sizeof(my_array) gives the total size of the array in bytes
    // sizeof(my_array[0]) gives the size of one element in bytes
    // Dividing them gives the number of elements
    int n = sizeof(my_array) / sizeof(my_array[0]);
    int search_target_1 = 60;
    int search_target_2 = 100;
    printf("Array elements: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", my_array[i]);
    }
    printf("\n");
    // Search for 60
    int index_1 = linearSearch(my_array, n, search_target_1);
    if (index_1 != -1) {
        printf("Element %d found at index: %d\n", search_target_1, index_1);
    } else {
```

```
        printf("Element %d not found in the array.\n", search_target_1);
    }
    // Search for 100
    int index_2 = linearSearch(my_array, n, search_target_2);
    if (index_2 != -1) {
        printf("Element %d found at index: %d\n", search_target_2, index_2);
    } else {
        printf("Element %d not found in the array.\n", search_target_2);
    }
    return 0; // Indicate successful execution
}
```

Linear Search vs. Binary Search

Both Linear Search and Binary Search are algorithms used to find a specific element within a collection of data (like an array). However, they differ significantly in their approach, efficiency, and requirements.

Linear Search

- Concept: Linear search is the simplest search algorithm. It sequentially checks each element of the array from the beginning until a match is found or the entire array has been traversed.
- Requirement: The array does not need to be sorted.
- Time Complexity:
    - Best Case: O(1) (if the target is the first element).
    - Worst Case: O(n) (if the target is the last element or not present).
    - Average Case: O(n/2), which simplifies to O(n).
    - This means the time taken increases linearly with the size of the input array.
- Space Complexity: O(1) (constant extra space).
- When to use: Suitable for small arrays or when the array is unsorted.

Example: Searching for 60 in [90, 50, 70, 10, 60, 30]

1. Compare 90 with 60. No match.
2. Compare 50 with 60. No match.
3. Compare 70 with 60. No match.

4. Compare 10 with 60. No match.
5. Compare 60 with 60. Match found at index 4. Return 4.

Binary Search

- Concept: Binary search is a much more efficient search algorithm, but it requires the data to be sorted. It works by repeatedly dividing the search interval in half. It compares the target value with the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated, and the search continues on the remaining half until the target is found or the interval becomes empty.
- Requirement: The array must be sorted in ascending or descending order.
- Time Complexity:
    - Best Case: O(1) (if the target is the middle element).
    - Worst Case: O(logn) (logarithmic time).
    - Average Case: O(logn).
    - This means the time taken increases logarithmically with the size of the input, making it very fast for large datasets.
- Space Complexity: O(1) (iterative approach) or O(logn) (recursive approach due to function call stack).
- When to use: Highly recommended for large, sorted arrays.

Example: Searching for 60 in [10, 30, 50, 60, 70, 90] (Sorted Array)

1. Initial range: low = 0, high = 5. mid = (0 + 5) / 2 = 2. Element at mid (index 2) is 50.
    - Since 60>50, the target is in the right half. New low = mid + 1 = 3.
    - Current range: low = 3, high = 5.
2. Next iteration: mid = (3 + 5) / 2 = 4. Element at mid (index 4) is 70.
    - Since 60<70, the target is in the left half. New high = mid - 1 = 3.
    - Current range: low = 3, high = 3.
3. Next iteration: mid = (3 + 3) / 2 = 3. Element at mid (index 3) is 60.
    - Since 60=60, match found at index 3. Return 3.

Key Differences Summarized

Feature	Linear Search	Binary Search
Requirement	No sorted order required	Must be sorted
Approach	Sequential, element by element	Divide and conquer, halves the search space
Efficiency	Less efficient (O(n))	More efficient (O(logn))
Use Case	Small arrays, unsorted data	Large arrays, sorted data
Comparisons	Many comparisons in worst case	Fewer comparisons, especially for large arrays

71. What is a function? Write all aspects of function with its syntax and an example.[1+4] 2. What is recursion? Explain with factorial example.

1. What is a Function?

A function in programming is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Aspects of a Function:

1. Purpose/Modularity:
    - Breaks down complex problems: Functions allow you to break down a large, complex problem into smaller, manageable sub-problems. Each function can solve a specific part of the problem.
    - Organizes code: They help in structuring the code logically, making it easier to read, understand, and maintain.
2. Reusability:
    - Once a function is defined, it can be called (executed) multiple times from different parts of the program without rewriting the same code. This saves development time and reduces errors.

3. Parameters (Arguments):
    - Functions can accept input values, known as parameters (or arguments), that allow them to operate on different data each time they are called. These parameters are placeholders for the values that will be passed into the function.
4. Return Value:
    - A function can process its inputs and then produce an output, known as a return value. This value is sent back to the part of the code that called the function. A function can return zero, one, or multiple values (though in C, a function typically returns a single value, or void if nothing is returned; multiple values can be simulated using pointers).
5. Scope:
    - Variables declared inside a function are typically local to that function, meaning they can only be accessed within that function. This prevents naming conflicts and unintended side effects between different parts of the program.
6. Abstraction:
    - Functions allow for abstraction, meaning you can use a function without knowing its internal implementation details. You only need to know what it does and how to use it (its interface).

Syntax (General C Language Example):

```
return_type function_name(parameter_type1 parameter_name1, parameter_type2
parameter_name2, ...) {
    // Function body:
    // Declarations
    // Statements
    // ...
    return value; // Optional: returns a value of return_type
}
```

- return_type: The data type of the value the function will return. If the function doesn't return any value, void is used.
- function_name: A unique identifier for the function.
- parameter_typeX parameter_nameX: A comma-separated list of parameters the function accepts. Each parameter has a type and a name. If a function takes no parameters, the parentheses can be empty or contain void.
- { ... }: The function body, containing the code that performs the function's task.

- return value;: An optional statement that sends a value back to the caller. The value must match the return_type.

Example (C Language):

```c
#include <stdio.h>

// Function definition: calculates the sum of two integers
int addNumbers(int a, int b) {
    int sum = a + b; // Calculate the sum
    return sum;      // Return the sum
}

int main() {
    int num1 = 10;
    int num2 = 20;
    int result;

    // Calling the function and storing its return value
    result = addNumbers(num1, num2);
    printf("The sum of %d and %d is: %d\n", num1, num2, result); // Output: The sum of 10 and 20 is: 30

    // Calling the function again with different arguments
    printf("The sum of 5 and 7 is: %d\n", addNumbers(5, 7)); // Output: The sum of 5 and 7 is: 12

    return 0;
}
```

In this example, addNumbers is a function that takes two integer parameters (a and b), calculates their sum, and returns an integer (sum). The main function calls addNumbers multiple times.

2. What is Recursion?

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. It's a way to define a function or a process in terms of itself. For a recursive function to work correctly, it must have:

1. Base Case: A condition that stops the recursion. Without a base case, the function would call itself indefinitely, leading to a stack overflow error.

2. Recursive Step: The part where the function calls itself, usually with a modified input that moves closer to the base case.

Recursion is often used to solve problems that can be broken down into smaller, similar sub-problems.

Explanation with Factorial Example:

The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n.

The definition is:

- $0! = 1$
- $n! = n \times (n-1)!$ for $n > 0$

This definition is inherently recursive, as n! is defined in terms of (n−1)!.

Let's see how this translates to a recursive function.

Recursive Factorial Function (Conceptual):

```
factorial(n):
    if n == 0:  // Base Case
        return 1
    else:       // Recursive Step
        return n * factorial(n - 1)
```

Example: Calculating 4! using Recursion

Let's trace the execution of factorial(4):

1. factorial(4) is called.
    - Is 4==0? No.
    - It executes return 4 * factorial(3).
    - Now, factorial(3) needs to be evaluated.
2. factorial(3) is called.
    - Is 3==0? No.
    - It executes return 3 * factorial(2).

    - Now, factorial(2) needs to be evaluated.
3. factorial(2) is called.
    - Is 2==0? No.
    - It executes return 2 * factorial(1).
    - Now, factorial(1) needs to be evaluated.
4. factorial(1) is called.
    - Is 1==0? No.
    - It executes return 1 * factorial(0).
    - Now, factorial(0) needs to be evaluated.
5. factorial(0) is called.
    - Is 0==0? Yes! (This is the Base Case)
    - It executes return 1.
    - The value 1 is returned to factorial(1).

Now, the calls start returning values back up the chain:

- factorial(1) receives 1 from factorial(0).
    - It computes 1 * 1 = 1.
    - The value 1 is returned to factorial(2).
- factorial(2) receives 1 from factorial(1).
    - It computes 2 * 1 = 2.
    - The value 2 is returned to factorial(3).
- factorial(3) receives 2 from factorial(2).
    - It computes 3 * 2 = 6.
    - The value 6 is returned to factorial(4).
- factorial(4) receives 6 from factorial(3).
    - It computes 4 * 6 = 24.
    - The value 24 is returned as the final result.

C Language Example for Factorial:

```c
#include <stdio.h>
// Recursive function to calculate factorial
long long factorial(int n) {
    // Base Case: If n is 0, return 1
    if (n == 0) {
        return 1;
    }
    // Recursive Step: n * factorial(n-1)
    else {
        return n * factorial(n - 1);
    }
}
int main() {
    int number = 4;
    long long result;
    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        result = factorial(number);
        printf("Factorial of %d is: %lld\n", number, result); // Output:
Factorial of 4 is: 24
    }
    int another_number = 0;
    result = factorial(another_number);
    printf("Factorial of %d is: %lld\n", another_number, result); // Output:
Factorial of 0 is: 1
    return 0;
}
```

This C code demonstrates the recursive implementation of the factorial function, clearly showing the base case and the recursive step.

72. Compare recursion and iteration with example.

Recursion vs. Iteration: A Comparison

Both recursion and iteration are fundamental programming constructs used to execute a set of instructions repeatedly. They achieve similar goals but use different approaches.

Recursion

Definition:

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. A problem is broken down into smaller, identical sub-problems, and the function solves these sub-problems by calling itself until a base case is reached.

Key Characteristics:

- Base Case: A condition that stops the recursion. Without it, the function would call itself infinitely, leading to a stack overflow.
- Recursive Step: The part where the function calls itself, usually with a modified input that moves closer to the base case.
- Stack Usage: Each recursive call adds a new frame to the call stack. This can lead to stack overflow errors for deep recursions.
- Elegance: Often leads to more concise and elegant code for problems that have a naturally recursive structure (e.g., tree traversals, certain mathematical sequences).

Pros:

- More readable and elegant for problems with inherent recursive definitions.
- Reduces code size for certain problems.
- Easier to write for problems that naturally lend themselves to recursive solutions.

Cons:

- Can be less efficient due to function call overhead (stack frame creation and destruction).
- Higher memory consumption due to the call stack.
- Risk of stack overflow for large inputs if the recursion depth is too high.
- Debugging can be more challenging due to the call stack.

Example: Factorial Calculation (Recursive)

The factorial of a non-negative integer n (n!) is defined as:

- 0!=1

- n!=n×(n−1)! for n>0

```c
#include <stdio.h>

// Recursive function to calculate factorial
long long factorial_recursive(int n) {
    // Base Case: stops the recursion
    if (n == 0) {
        return 1;
    }
    // Recursive Step: calls itself with a smaller input
    else {
        return n * factorial_recursive(n - 1);
    }
}
```

Iteration

Definition:

Iteration is a programming technique where a set of instructions is repeated a specific number of times or until a certain condition is met. This is typically achieved using loops (e.g., for loop, while loop, do-while loop).

Key Characteristics:

- Loop Control: Uses a loop construct (like for, while) with a counter or a condition to control repetitions.
- Explicit State: The state of the computation (e.g., loop counter, accumulated sum) is explicitly managed by variables.
- No Stack Overhead: Does not involve function call overhead, making it generally more efficient in terms of time and memory.

Pros:

- Generally more efficient in terms of time and memory (no function call overhead).
- Easier to analyze and optimize performance.

- Less prone to stack overflow errors.
- Debugging is often simpler as the flow of control is more linear.

Cons:

- Can be less intuitive or more verbose for problems that have a natural recursive structure.
- Might require more code to manage loop variables and conditions.

Example: Factorial Calculation (Iterative)

```c
#include <stdio.h>

// Iterative function to calculate factorial
long long factorial_iterative(int n) {
    long long result = 1; // Initialize result
    // Handle base case for 0!
    if (n == 0) {
        return 1;
    }
    // Loop from 1 up to n, multiplying result by each number
    for (int i = 1; i <= n; i++) {
        result *= i; // result = result * i
    }
    return result;
}
```

Comparison Table

Feature	Recursion	Iteration
Approach	Function calls itself	Uses loops (for, while, do-while)
Control Flow	Managed by function calls and return values	Managed by loop conditions and counters
Memory	Higher (uses call stack for each call)	Lower (constant memory usage)

Speed	Slower (due to function call overhead)	Faster (no function call overhead)
Code Size	Often more concise and elegant	Can be more verbose for complex problems
Debugging	Can be harder (tracing call stack)	Generally easier (linear flow)
Stack Overflow	Risk for deep recursion	Not a risk for typical loop sizes
Base Case/Termination	Base case is essential for termination	Loop condition is essential for termination

When to Choose Which

- Choose Recursion when:
    - The problem has a natural recursive definition (e.g., tree traversals, graph algorithms, divide-and-conquer algorithms like quicksort/mergesort).
    - Code clarity and elegance are prioritized, and the problem size is not excessively large (to avoid stack overflow).
- Choose Iteration when:
    - Performance and memory efficiency are critical.
    - The problem can be easily expressed using loops.
    - The problem size might lead to deep recursion, risking stack overflow.

In many cases, a problem can be solved using both recursion and iteration. The choice often comes down to readability, performance requirements, and the specific nature of the problem.