CMR
INSTITUTE OF TECHNOLOGY

IAT2 – MARCH 2025
I SEM MCA
Programming and Problem Solving in C-MMC101
SOLUTION

	PART I	MARKS	OBE	
			CO	RBT
1	Explain the concept of recursion. Write a C program to find the GCD (Greatest Common Divisor) of two numbers using recursion. **OR**	[10]	CO3	L3
2	Implement binary search using recursion and explain how it works with an example.	[10]	CO3	L3
	PART II			
3	Differentiate between pass by value and pass by reference with an example. **OR**	[10]	CO3	L2
4	What is an array of pointers? Write a C program to store and display student names using an array of pointers.	[10]	CO3	L4
	PART III			
5	Define a structure. How can a structure be passed to a function? Demonstrate with a program. **OR**	[10]	CO4	L4
6	a. Demonstrate nested structures with an example. b. What are self-referential structures? Give an example.	[5 + 5]	CO4	L4
	PART IV			
7	Explain different storage classes in C with examples. **OR**	[10]	CO5	L3
8	Write a C program to create a singly linked list and display its elements.	[10]	CO4	L5
	PARTV			
9	Explain dynamic memory allocation functions in C with examples. **OR**	[10]	CO4	L3
10	What are the different modes of opening a file in C. Explain with examples?	[10]	CO5	L3

SOLUTION

1. **Explain the concept of recursion. Write a C program to find the GCD (Greatest Common Divisor) of two numbers using recursion.**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same

problem. It consists of:

1. Base Case – The condition where recursion stops.
2. Recursive Case – The function calls itself with modified arguments.

Recursion is useful for problems like factorial, Fibonacci series, GCD, and tree-based algorithms.

```
#include <stdio.h>

// Recursive function to find GCD
int gcd(int a, int b) {
    if (b == 0)  // Base case: If second number becomes 0, return the first number
        return a;
    return gcd(b, a % b);  // Recursive case: GCD(b, remainder of a divided by b)
}

int main() {
    int num1, num2;

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

    // Call recursive GCD function and display result
    printf("GCD of %d and %d is: %d\n", num1, num2, gcd(num1, num2));

    return 0;
}
```

2. Implement binary search using recursion and explain how it works with an example.

Binary search is an efficient algorithm to search for an element in a sorted array. It works by:

1. Finding the middle element of the array.
2. If the middle element matches the key, return its index.
3. If the key is smaller, search in the left half.
4. If the key is larger, search in the right half.
5. Repeat until the key is found or the search space is exhausted.

```
#include <stdio.h>

// Recursive function for binary search
int binarySearch(int arr[], int low, int high, int key) {
    if (low > high)  // Base case: Element not found
        return -1;

    int mid = low + (high - low) / 2; // Calculate middle index

    // Check if the key is at the middle
    if (arr[mid] == key)
        return mid;

    // Search in the left half
    if (key < arr[mid])
        return binarySearch(arr, low, mid - 1, key);
```

```
    // Search in the right half
    return binarySearch(arr, mid + 1, high, key);
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}; // Sorted array
    int n = sizeof(arr) / sizeof(arr[0]);
    int key;

    // Get input from user
    printf("Enter the element to search: ");
    scanf("%d", &key);

    // Call binary search
    int result = binarySearch(arr, 0, n - 1, key);

    // Print result
    if (result != -1)
        printf("Element %d found at index %d\n", key, result);
    else
        printf("Element %d not found in the array\n", key);

    return 0;
}
```

3. Differentiate between pass by value and pass by reference with an example.

Feature	Pass by Value	Pass by Reference
Definition	A copy of the actual value is passed to the function.	The address of the actual variable is passed.
Effect on Original Variable	The original variable remains unchanged.	The original variable can be modified.
Memory Usage	More memory is used as a new copy is created.	Less memory is used as only an address is passed.
Function Call Overhead	Higher, as new memory is allocated.	Lower, as no new copy is created.
Use Case	Used when modifications to the original data are not needed.	Used when modifications to the original data are required.

Example of Pass by Value

```
#include <stdio.h>

// Function to swap two numbers (Pass by Value)
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    printf("Inside function (after swap): a = %d, b = %d\n", a, b);
```

```
}

int main() {
   int x = 10, y = 20;
   printf("Before function call: x = %d, y = %d\n", x, y);
   swap(x, y);
   printf("After function call: x = %d, y = %d\n", x, y); // Values remain unchanged
   return 0;
}
```

Output:
Before function call: x = 10, y = 20
Inside function (after swap): a = 20, b = 10
After function call: x = 10, y = 20

The swap function changes a and b, but x and y remain unchanged.

Example of Pass by Reference

```
#include <stdio.h>

// Function to swap two numbers (Pass by Reference)
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int x = 10, y = 20;
   printf("Before function call: x = %d, y = %d\n", x, y);
   swap(&x, &y);
   printf("After function call: x = %d, y = %d\n", x, y); // Values are swapped
   return 0;
}
```

Output:
Before function call: x = 10, y = 20
After function call: x = 20, y = 10

Since the function receives addresses (&x and &y), it directly modifies x and y.

4. What is an array of pointers? Write a C program to store and display student names using an array of pointers.

An array of pointers is a collection of memory addresses, where each element of the array stores the address of another variable or object (such as a string).
When dealing with strings in C, an array of pointers is more memory-efficient than a 2D character array because it stores only the addresses of the strings instead of allocating a fixed-length buffer for each.

```
#include <stdio.h>

int main() {
   // Array of pointers to strings (student names)
```

```
    char *students[] = {
        "Alice",
        "Bob",
        "Charlie",
        "David",
        "Eve"
    };

    int n = sizeof(students) / sizeof(students[0]); // Number of students

    // Display the student names
    printf("List of students:\n");
    for (int i = 0; i < n; i++) {
        printf("%s\n", students[i]);  // Accessing string using pointer array
    }

    return 0;
}
```

5. Define a structure. How can a structure be passed to a function? Demonstrate with a program.

A structure in C is a user-defined data type that allows grouping different types of variables under a single name. It is useful for creating complex data models like student records, employee details, etc.

A structure can be passed to a function in two ways:

1. **Pass by Value** – A copy of the structure is passed (original remains unchanged).
2. **Pass by Reference (Using Pointers)** – The address of the structure is passed (original can be modified).

```
#include <stdio.h>

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

// Function to display student details (Pass by Value)
void displayStudent(struct Student s) {
    printf("\nStudent Details:\n");
    printf("Name: %s\n", s.name);
    printf("Roll Number: %d\n", s.rollNo);
    printf("Marks: %.2f\n", s.marks);
}

// Function to modify student marks (Pass by Reference)
void updateMarks(struct Student *s, float newMarks) {
    s->marks = newMarks; // Modify original structure using pointer
}
```

```
int main() {
    struct Student student1;

    // Input student details
    printf("Enter Student Name: ");
    scanf("%s", student1.name);
    printf("Enter Roll Number: ");
    scanf("%d", &student1.rollNo);
    printf("Enter Marks: ");
    scanf("%f", &student1.marks);

    // Call function (Pass by Value)
    displayStudent(student1);

    // Modify marks using Pass by Reference
    updateMarks(&student1, 95.0);
    printf("\nAfter Updating Marks:\n");
    displayStudent(student1);

    return 0;
}
```

6. Demonstrate nested structures with an example.
What are self-referential structures? Give an example.

A nested structure is a structure within another structure. It helps in organizing complex data hierarchically.

```
#include <stdio.h>

// Defining a nested structure
struct Address {
    char city[50];
    char state[50];
    int pinCode;
};

struct Student {
    char name[50];
    int rollNo;
    struct Address addr; // Nested structure
};

int main() {
    struct Student s1;

    // Input student details
    printf("Enter Student Name: ");
    scanf("%s", s1.name);
    printf("Enter Roll Number: ");
    scanf("%d", &s1.rollNo);
    printf("Enter City: ");
    scanf("%s", s1.addr.city);
```

```
    printf("Enter State: ");
    scanf("%s", s1.addr.state);
    printf("Enter Pincode: ");
    scanf("%d", &s1.addr.pinCode);

    // Display student details
    printf("\nStudent Details:\n");
    printf("Name: %s\nRoll Number: %d\n", s1.name, s1.rollNo);
    printf("City: %s\nState: %s\nPincode: %d\n", s1.addr.city, s1.addr.state, s1.addr.pinCode);

    return 0;
}
```

A self-referential structure is a structure that contains a pointer to itself. It is commonly used for linked lists, trees, and other dynamic data structures.

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

// Defining a self-referential structure (Linked List Node)
struct Node {
    int data;
    struct Node *next; // Pointer to another Node
};

int main() {
    struct Node *head = NULL, *second = NULL, *third = NULL;

    // Allocating memory for nodes
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));

    // Assigning values and linking nodes
    head->data = 10;
    head->next = second;

    second->data = 20;
    second->next = third;

    third->data = 30;
    third->next = NULL; // End of the list

    // Display the linked list
    struct Node *temp = head;
    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");

    // Free allocated memory
    free(head);
    free(second);
```

```
    free(third);

    return 0;
}
```

7. Explain different storage classes in C with examples.

Storage Classes in C

In C, a storage class defines the scope (visibility), lifetime, and memory location of a variable. The four main storage classes are:

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

1. Automatic Storage Class (auto)

- Default storage class for local variables.
- Scope: Local (inside the function/block where declared).
- Lifetime: Created when the function starts and destroyed when the function ends.
- Memory Location: Stored in stack.
- Keyword: auto (optional, rarely used).

Example:

c

CopyEdit

```
#include <stdio.h>

void func() {
    auto int x = 10; // 'auto' is optional
    printf("Auto Variable x = %d\n", x);
}

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

Output:

java

CopyEdit

Auto Variable x = 10

Since x is local to func(), it is created and destroyed each time func() is called.

2. Register Storage Class (register)

- Similar to auto, but suggests storing the variable in CPU registers (for faster access).
- Scope: Local to the function/block.
- Lifetime: Created and destroyed with the function.
- Memory Location: CPU registers (if available, else stored in RAM).
- Keyword: register.
- Limitation: Cannot use & (address-of operator) as it may not have a memory address.

Example:

c

CopyEdit

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

```
int main() {
    register int i = 5;
    printf("Register Variable i = %d\n", i);
    return 0;
}
```

Output:

java

CopyEdit

Register Variable i = 5

The compiler may store i in a CPU register for faster access.

3. Static Storage Class (static)

- Scope: Local or global (depends on where declared).
- Lifetime: Exists throughout the program execution.
- Memory Location: Global (Data) Segment.
- Keyword: static.
- Used for preserving values across function calls.

Example (Static Local Variable):

c

CopyEdit

```
#include <stdio.h>

void counter() {
    static int count = 0; // Retains value between function calls
    count++;
    printf("Count = %d\n", count);
}

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

Output:

ini

CopyEdit

Count = 1

Count = 2

Count = 3

Unlike auto, static retains count value between function calls.

Example (Static Global Variable):

c

CopyEdit

```
#include <stdio.h>

static int globalVar = 100; // Accessible within this file only

void func() {
    printf("Static Global Variable = %d\n", globalVar);
```

```
}

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

Output:

sql
CopyEdit
Static Global Variable = 100
A static global variable is not accessible from other files.

4. External Storage Class (extern)

- Scope: Global (accessible across multiple files).
- Lifetime: Exists throughout program execution.
- Memory Location: Global (Data) Segment.
- Keyword: extern.
- Used for global variables defined in another file.

Example (Using extern Across Files)
File1.c (Define global variable)
c
CopyEdit

```
#include <stdio.h>

int num = 50; // Global variable (definition)

void display() {
   printf("Value of num in File1: %d\n", num);
}
```

File2.c (Access global variable using extern)
c
CopyEdit

```
#include <stdio.h>

extern int num; // Declaration of external variable

int main() {
   printf("Value of num in File2: %d\n", num);
   return 0;
}
```

Output (After Compiling Both Files):
yaml
CopyEdit
Value of num in File2: 50

Extern allows num to be accessed across multiple files.

8. Write a C program to create a singly linked list and display its elements.

A singly linked list consists of nodes, where each node contains:

1. Data (stores the value)

2. Pointer (next) to the next node in the list.
The last node's next pointer is NULL, indicating the end of the list.

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

// Define a structure for a node
struct Node {
   int data;
   struct Node *next; // Pointer to the next node
};

// Function to create a linked list
struct Node* createLinkedList(int n) {
   struct Node *head = NULL, *temp = NULL, *newNode = NULL;
   int value, i;

   for (i = 1; i <= n; i++) {
      newNode = (struct Node*)malloc(sizeof(struct Node)); // Allocate memory
      if (newNode == NULL) {
         printf("Memory allocation failed!\n");
         return NULL;
      }

      printf("Enter data for node %d: ", i);
      scanf("%d", &value);

      newNode->data = value;
      newNode->next = NULL;

      if (head == NULL) {
         head = newNode; // First node becomes head
         temp = head;
      } else {
         temp->next = newNode; // Link previous node to new node
         temp = newNode; // Move temp to the new node
      }
   }
   return head;
}

// Function to display the linked list
void displayLinkedList(struct Node *head) {
   struct Node *temp = head;

   if (head == NULL) {
      printf("The linked list is empty.\n");
      return;
   }

   printf("\nLinked List Elements: ");
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
```

```
}

// Main function
int main() {
    struct Node *head = NULL;
    int n;

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

    head = createLinkedList(n); // Create the linked list
    displayLinkedList(head); // Display the linked list

    return 0;
}
```

9. Explain dynamic memory allocation functions in C with examples.

Dynamic memory allocation (DMA) allows us to allocate memory at runtime rather than at compile time. This helps when we don't know the exact amount of memory required beforehand.
C provides four built-in functions for dynamic memory allocation, all defined in stdlib.h:

1. malloc() – Allocates memory but does not initialize it.
2. calloc() – Allocates and initializes memory to zero.
3. realloc() – Resizes previously allocated memory.
4. free() – Deallocates memory.

1. malloc() (Memory Allocation)

- Allocates uninitialized memory.
- Returns a pointer to the first byte of the allocated memory.
- Returns NULL if memory allocation fails.

Syntax:

```
ptr = (cast_type*) malloc(size_in_bytes);

    ptr = (int*) malloc(n * sizeof(int)); // Allocating memory for 5 integers
```

2. calloc() (Contiguous Allocation)

- Allocates multiple blocks of memory and initializes all bytes to zero.
- Useful when you want zero-initialized memory.

Syntax:

```
ptr = (cast_type*) calloc(num_elements, element_size);

    ptr = (int*) calloc(n, sizeof(int)); // Allocating memory for 5 integers
```

3. realloc() (Reallocation)

- Resizes previously allocated memory.
- Used when we need more or less memory than initially allocated.

Syntax:

```
ptr = (cast_type*) realloc(ptr, new_size);
```

Example:

```
    ptr = (int*) malloc(n * sizeof(int));

    // Expanding memory to store 5 integers
```

```
n = 5;
ptr = (int*) realloc(ptr, n * sizeof(int));
```

4. free() (Memory Deallocation)

- Frees dynamically allocated memory to prevent memory leaks.
- Does not delete the pointer, only releases the memory.

Example:

```
free(ptr); // Free allocated memory
```

10. What are the different modes of opening a file in C. Explain with examples?

C provides file handling capabilities through the stdio.h library, which allows opening, reading, writing, and closing files.
To open a file, we use:
c
CopyEdit
FILE *fp;
fp = fopen("filename", "mode");
filename → Name of the file.
mode → Specifies how the file will be opened.

Mode	Description
r	**opens a text file in readingmode**
w	**opens or create a text file in writing mode.**
a	**opens a text file in appendmode**
r+	**opens a text file in both reading and writing mode**
w+	**opens a text file in both reading and writing mode**
a+	**opens a text file in both reading and writing mode**
rb	**opens a binary file in reading mode**
wb	**opens or create a binary file in writing mode**
ab	**opens a binary file in append mode**
rb+	**opens a binary file in both reading and writing mode**
wb+	**opens a binary file in both reading and writing mode**
ab+	**opens a binary file in both reading and writing mode**