CMR
INSTITUTE OF TECHNOLOGY

Department of MCA

Programming & Problem Solving Using C- MMC101
I MCA IAT2 – Feb 2026
SOLUTION

	PART I	MARKS	OBE	
			CO	RBT
1	Define Recursion. Write a program to implement binary search using recursion. **OR**	[10]	CO3	L3
2	Write a c program to find the Largest Element in an Array using Pointers	[10]	CO3	L3
	PART II			
3	Differentiate between Call by value and Call by reference with example. **OR**	[10]	CO3	L2
4	Discuss the advantages of using pointers in C. Explain pointer arithmetic with suitable examples.	[10]	CO3	L4
	PART III			
5	Define a structure. Write a C program to store and display student details using structures. **OR**	[10]	CO4	L3
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	Define Linked List and perform insertion of a node at the beginning of linked list.	[10]	CO4	L5
	PARTV			
9	Explain dynamic memory allocation functions in C with examples. **OR**	[10]	CO4	L3
10	Write a C program to copy the contents of one file to another.	[10]	CO5	L4

SOLUTION

1. Define Recursion. Write a program to implement binary search using recursion.

1. Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It consists of
2. Base Case – The condition where recursion stops.
3. Recursive Case – The function calls itself with modified arguments.

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

2. Write a C program to find the Largest Element **in an Array using Pointers**

```
#include <stdio.h>

int main()
{
    int n, i;
    int arr[100];
    int *ptr;
    int largest;

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

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

    /* Initialize pointer to first element */
    ptr = arr;

    /* Assume first element is largest */
    largest = *ptr;

    /* Traverse array using pointer */
    for(i = 1; i < n; i++)
    {
        ptr++;
        if(*ptr > largest)
        {
            largest = *ptr;
        }
    }

    printf("Largest element = %d\n", largest);

    return 0;
}
```

3. Differentiate between Cass by value and Cass 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 Cass 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. Discuss the advantages of using pointers in C. Explain pointer arithmetic with suitable examples.

Advantages of Using Pointers in C

1. **Efficient memory access:** Pointers allow direct access to memory locations, making programs faster.
2. **Call by reference:** Pointers allow functions to modify original variables.
3. **Dynamic memory allocation:** Used with functions like malloc() and free() to allocate memory at runtime.
4. **Efficient array and string handling:** Arrays and strings can be easily accessed and manipulated using pointers.
5. **Used in data structures:** Pointers are essential for linked lists, trees, and other dynamic structures.

Pointer Arithmetic

Pointer arithmetic means performing operations like increment, decrement, addition, and subtraction on pointers.

Increment Operator (++)

Moves the pointer to the next memory location of its data type.

Example:

```
int arr[] = {10, 20, 30};
int *p = arr;

p++;  // moves to next element
```

2. Decrement Operator (--)

Moves the pointer to the previous memory location.

Example:

```
p--;  // moves to previous element
```

---

3. Addition (+)

Adds an integer value to a pointer to move forward multiple elements.

Example:

```
int *p = arr;
p = p + 2;  // moves 2 elements forward
```

---

4. Subtraction (-) with integer

Subtracts an integer value from a pointer to move backward.

Example:

```
p = p - 1;  // moves 1 element backward
```

5. Subtraction between two pointers

Finds the number of elements between two pointers.

Example:

```
int *p = &arr[3];
int *q = &arr[0];

int diff = p - q;  // gives number of elements between them
```

6. Comparison of pointers

Pointers can be compared using relational operators.

Example:

```
if(p > q)
   printf("p points to higher address");
```

Operations NOT allowed on pointers

These operations are invalid:

- Multiplication (*)
- Division (/)
- Modulus (%)
- Addition of two pointers

5. **Define a structure. Write a C program to store and display student details using structures.**

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.

```
#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);
}
```

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:

Count = 1

Count = 2

Count = 3

Unlike auto, static retains count value between function calls.

Example (Static Global Variable):

```
#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:

```
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)

```
#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

```
#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):

```
Value of num in File2: 50
```

Extern allows num to be accessed across multiple files.

8. Define Linked List and perform insertion of a node at the beginning of linked list.

A **Linked List** is a dynamic data structure consisting of a collection of nodes, where each node contains two parts:

1. Data part – stores the value
2. Link part – stores the address of the next node

Nodes are connected using pointers.

Unlike arrays, linked lists do not use contiguous memory locations.

Insertion at the Beginning of Linked List

Insertion at the beginning means adding a new node before the first node.

Steps:

1. Create a new node
2. Store data in the new node
3. Make the new node point to the current first node
4. Update head pointer to the new node.

```
    // Define a node structure
    struct Node {
        int data;
        struct Node* next;
    };

struct Node* head = NULL;

       void insertAtBeginning() {
           struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
           if (newNode==NULL) {
               printf("Memory allocation failed!\n");
               return;
           }
           printf("Enter data: ");
           scanf("%d", &newNode->data);
           newNode->next = head;
           head = newNode;
           printf("Node inserted at beginning.\n");
       }
```

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. Write a C program to copy the contents of one file to another.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
FILE *fp1, *fp2, *fp3;
char file1[50], file2[50], outputFile[50], ch;
// Get filenames from user
printf("Enter first file name: ");
scanf("%s", file1);
printf("Enter second file name: ");
scanf("%s", file2);
printf("Enter output file name: ");
scanf("%s", outputFile);
// Open first file
fp1 = fopen(file1, "r");
if (fp1 == NULL) {
printf("Error opening %s\n", file1);
exit(1);
}
// Open second file
fp2 = fopen(file2, "r");
if (fp2 == NULL) {
printf("Error opening %s\n", file2);
fclose(fp1);
exit(1);
}
// Open third file in write mode
fp3 = fopen(outputFile, "w");
if (fp3 == NULL) {
printf("Error opening %s\n", outputFile);
fclose(fp1);
fclose(fp2);
exit(1);
}
// Copy contents of first file to output file
while ((ch = fgetc(fp1)) != EOF) {
fputc(ch, fp3);
}
fclose(fp1); // Close first file
// Copy contents of second file to output file
while ((ch = fgetc(fp2)) != EOF) {
fputc(ch, fp3);
}
fclose(fp2); // Close second file
fclose(fp3); // Close output file
printf("Files merged successfully into %s\n", outputFile);
return 0;}
```