USN 1CR25MC044

MMC101

First Semester MCA Degree Examination, Dec.2025/Jan.2026

Programming and Problem Solving in C

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain the structure of C program.	06	L1	CO1
	b.	Explain the uses of C language.	08	L1	CO1
	c.	Write the key words of C language.	06	L2	CO1
		OR			
Q.2	a.	Explain conditional statements with examples.	12	L2	CO1
	b.	Explain with example loops in C programming.	08	L2	CO1
		Module – 2			
Q.3	a.	Differentiate between 1-D and 2 D arrays. Explain with example.	10	L2	CO2
	b.	Write C program to multiply two matrices& display the result in transpose form.	10	L2	CO3
		OR			
Q.4	a.	Define string. Explain taxonomy of the string.	08	L2	CO3
	b.	Explain the operations on string.	05	L3	CO3
	c.	Write a C program search element in an array using linear search.	07	L3	CO3
		Module – 3			
Q.5	a.	Define a Function. Differentiate between call by value and call by reference.	08	L2	CO4
	b.	Write a C program to generate the Fibonacci numbers using recursive function.	07	L3	CO4
	c.	Write a C program to swap two numbers without using temp variable.	05	L3	CO4
		OR			
Q.6	a.	Write a C program to find the mean of N numbers using arrays and pointers	10	L3	CO3,CO2
	b.	Write a C program to add two matrices using pointers.	10	L3	CO3
		Module – 4			
Q.7	a.	Define Structure. Give syntax of Structure.	05	L2	CO5
	b.	Write a C program using structure to read and display student information.	10	L3	CO5
	c.	Explain nested structure with example	05	L2	CO5
		OR			
Q.8	a.	Define Union. Explain the syntax of Union with example.	05	L2	CO5
	b.	Explain the various storage classes	10	L3	CO5
	c.	Write a short note typedef	05	L2	CO5
		Module – 5			
Q.9	a.	Define a File. List and explain the operations on file	10	L2	CO1
	b.	List and describe the file modes.	10	L2	CO1
		OR			
Q.10	a.	List and explain functions for Reading strings.	10	L2	CO1
	b.	List and explain functions for Writing strings.	10	L2	CO1

* * * * *

1st Sem MCA Examination Dec 2025./Jan 2026
Programming & Problem Solving using C – MMC101

Solution

Module 1

Q1 (a) Explain the structure of a C program.

Answer:
A typical C program has the following structure:

```
Documentation Section
Pre-processor directives
Definition Section and Global declarations void
main()
{
        Declaration part

        Executable part

}
Sub Program Section
{
        Body of the Sub

}
```

1. Preprocessor directives (e.g., #include <stdio.h>)
2. Global declarations
3. main() function – the execution starts here
4. Function definitions

Example:

```
#include <stdio.h>  // Header file
int main() {
   printf("Structure Example");
   return 0;
}
```

Q1 (b) Explain the uses of C Language.

Answer:
C is used in various domains:

The C programming language is one of the most widely used languages in the world. Its power, efficiency, and closeness to hardware make it ideal for many domains. Below are the key uses of C language, explained in detail:

1. System Programming - C is used to develop operating systems, system tools, and device drivers. It provides low-level access to memory using pointers and direct hardware manipulation.**Example:** UNIX and Linux kernels are written in C.

2. Embedded Systems- Embedded software in devices like washing machines, microwaves, smart TVs, routers, etc., are often written in C. C is preferred because of its speed, portability, and direct control over hardware.

3. Game Development-C provides high performance and real-time memory management which is essential for game engines.

4. Compiler Design- Many compilers for other programming languages (like C++, Java, etc.) are implemented using C. Its efficiency and control make it ideal for writing language parsers and interpreters.

5. Database Systems - Popular databases like MySQL and Oracle use C in their core code.C helps manage memory and database operations effectively.

6. Operating Systems- Many modern OS like Linux, Windows (some parts), and macOS include code written in C. It is used to implement the core system components, schedulers, and memory management.

7. Network Drivers and Protocols - C is widely used in developing networking components like protocols (TCP/IP), routers, and switch firmware.

8. GUI (Graphical User Interface) Applications -Though high-level languages are more common now, C is still used for building basic GUI applications with toolkits like GTK.

9. Competing in Competitive Programming - Due to its simple syntax and fast execution, C is often used in coding contests and interviews.

10. Portability - Programs written in C can be compiled and run on different machines with little or no modification, making it ideal for cross-platform development.

11. Education and Learning

Q1 (c) Write the keywords of C language.

Answer:

Keywords are reserved words with predefined meanings.

Examples:

- Data types: `int, float, char`
- Control: `if, else, switch`
- Loops: `for, while, do`
- Others: `return, break, continue`

Total keywords in C = **32**

Q2 (a) Explain conditional statements with examples.

Answer:

Conditional statements in C are used to **control the flow of execution** based on certain conditions. They allow the program to make decisions.

Types of Conditional Statements:

1. Simple if Statement

This is the basic form where a condition is tested. If the condition is true, the statement inside the if block is executed.

Syntax:

```
if (condition) {
   // code to execute if condition is true
}
```

Example:

```
int a = 10;
if (a > 0) {
   printf("a is positive");
}
```

2. if-else Statement

This form allows two paths: one if the condition is true, and another if it is false.

Syntax:

```
if (condition) {
   // code if condition is true
} else {
   // code if condition is false
}
```

Example:

```
int a = -5;
if (a > 0) {
   printf("Positive number");
} else {
   printf("Non-positive number");
}
```

3. if-else if-else Ladder

Used to check multiple conditions sequentially. As soon as one condition is true, the corresponding block is executed.

Syntax:

```
if (condition1) {
   // code block 1
} else if (condition2) {
   // code block 2
} else {
   // default code block
}
```

Example:

```
int marks = 75;
if (marks >= 90) {
   printf("Grade A");
} else if (marks >= 75) {
   printf("Grade B");
} else {
   printf("Grade C");
}
```

4. Nested if Statement

An if statement inside another if block is known as a nested if. This is used for checking multiple related conditions.

Syntax:

```
if (condition1) {
   if (condition2) {
      // code to execute
   }
}
```

Example:

```
int age = 25;
int citizen = 1;
if (age >= 18) {
   if (citizen == 1) {
      printf("Eligible to vote");
   }
}
```

5. switch Statement

- Used for multiple choices based on value

Syntax:

```
switch(expression) {
   case value1: statement; break;
   case value2: statement; break;
   default: statement;
}
```

Example:

```
#include<stdio.h>
int main() {
   int day = 2;

   switch(day) {
      case 1: printf("Monday"); break;
      case 2: printf("Tuesday"); break;
      default: printf("Invalid");
   }
}
```

Q2 (b) Explain with example loops in C programming.

Answer:

In C programming, loops are used to repeatedly execute a block of code based on a condition. There are three primary types of loops: while, do-while, and for loops.

1. While Loop

The while loop executes a block of code as long as the given condition is true.

Syntax:

```
while (condition) {
   // Code to be executed
}
```

Example:

```
#include <stdio.h>

int main() {
   int i = 1;
   while (i <= 5) {
      printf("%d\n", i);
      i++;
   }
   return 0;
}
```

Output:

```
1
2
3
4
5
```

2. Do-While Loop

The do-while loop is similar to the while loop, but the condition is checked **after** the block of code is executed. This means the loop will always execute at least once.

Syntax:

```
do {
   // Code to be executed
} while (condition);
```

Example:

```
#include <stdio.h>

int main() {
   int i = 1;
   do {
      printf("%d\n", i);
      i++;
   } while (i <= 5);
   return 0;
}
```

Output:

```
1
2
3
```

4
5

3. For Loop

The for loop is typically used when the number of iterations is known beforehand. It combines initialization, condition checking, and increment/decrement in a single line.

Syntax:

```
for (initialization; condition; increment/decrement) {
   // Code to be executed
}
```

Example:

```
#include <stdio.h>

int main() {
   for (int i = 1; i <= 5; i++) {
      printf("%d\n", i);
   }
   return 0;
}
```

Output:

1
2
3
4
5

Module 2

Q3 (a) Differentiate between 1-D and 2D array. Explain with example.

Answer:
An **array** is a collection of elements of the same data type stored in **contiguous memory locations**. Arrays are used to store multiple values under a single variable name.

Based on dimensions, arrays are classified into:

- **One-Dimensional (1-D) Array**
- **Two-Dimensional (2-D) Array**

1. One-Dimensional (1-D) Array

Definition:

A 1-D array is a **linear list of elements** stored in a single row.

Syntax:

```
int a[5];
```

Example:

```
int a[5] = {10, 20, 30, 40, 50};
```

Accessing Elements:

- Using a single index

```
a[0], a[1], a[2]
```

Two-Dimensional (2-D) Array

Definition:

A 2-D array is a collection of elements arranged in rows and columns (matrix form).

Syntax:
int a[3][3];

Example:

```
int a[2][2] = {
   {1, 2},
   {3, 4}
};
```

Accessing Elements:

- Using two indices

a[0][0], a[1][1]

Feature	**1-D Array**	**2-D Array**
Definition	Linear collection	Matrix form
Dimensions	One dimension	Two dimensions
Syntax	`int a[5]`	`int a[3][3]`
Access	Single index `a[i]`	Double index `a[i][j]`
Representation	Single row	Rows and columns
Usage	List of values	Tables, matrices
Memory	Linear	Row-major order

Q3 (b)Write a C program to multiply 2 matrices and give result in transpose form.

Matrix Multiplication Program:

Multiplication of two matrices is done by multiplying corresponding elements from the rows of the first matrix with the corresponding elements from the columns of the second matrix and then adding these products. The number of columns in the first matrix must be equal to the number of rows in the second matrix.

Program

```
#include <stdio.h>

int main() {
   int a[10][10], b[10][10], c[10][10], t[10][10];
   int i, j, k, r1, c1, r2, c2;

   // Input dimensions
   printf("Enter rows and columns of matrix A: ");
   scanf("%d%d", &r1, &c1);

   printf("Enter rows and columns of matrix B: ");
   scanf("%d%d", &r2, &c2);

   // Check multiplication condition
   if (c1 != r2) {
      printf("Multiplication not possible\n");
      return 0;
   }

   // Input Matrix A
   printf("Enter elements of Matrix A:\n");
   for (i = 0; i < r1; i++)
      for (j = 0; j < c1; j++)
         scanf("%d", &a[i][j]);

   // Input Matrix B
   printf("Enter elements of Matrix B:\n");
   for (i = 0; i < r2; i++)
      for (j = 0; j < c2; j++)
         scanf("%d", &b[i][j]);

   // Matrix Multiplication
   for (i = 0; i < r1; i++) {
      for (j = 0; j < c2; j++) {
         c[i][j] = 0;
         for (k = 0; k < c1; k++) {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
   }
```

```
    // Transpose of Result Matrix
    for (i = 0; i < r1; i++) {
        for (j = 0; j < c2; j++) {
            t[j][i] = c[i][j];
        }
    }

    // Display Transpose Matrix
    printf("Transpose of Result Matrix:\n");
    for (i = 0; i < c2; i++) {
        for (j = 0; j < r1; j++) {
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Q4 (a) Define string. Explain taxonomy of string.

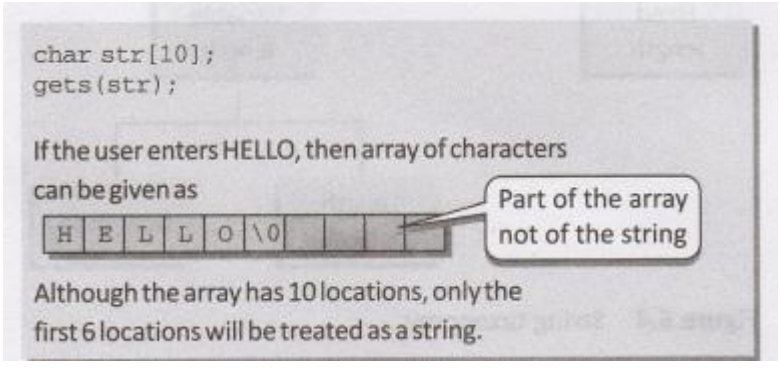
Answer:
String is a sequence of characters ending with a null character \0.

Example:
char name[] = "CMRIT";

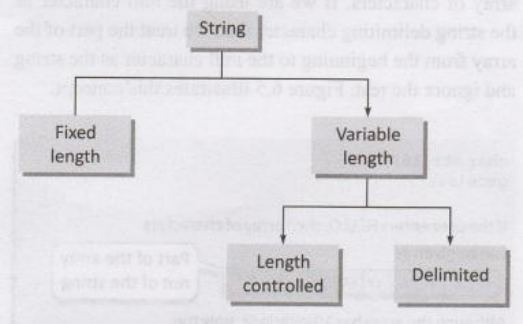
Taxonomy:
In C programming, string taxonomy refers to the classification and representation of strings.

String Representation:

- Character Array: A sequence of characters stored in an array, terminated by '\0' (null character).
- String Literal: A fixed sequence of characters stored in read-only memory.



In C, we can store a string either in fixed-length format or in variable-length format:



- **Fixed-length string** When storing a string in a fixed- length format, you need to specify an appropriate size for the string variable. If the size is too small, then you will not be able to store all the elements in the string. On the other hand, if the string size is large, then unnecessarily memory space will be wasted.
- **Variable-length string** A better option is to use a variable length format in which the string can be expanded or contracted to accommodate the elements in it. For example, if you declare a string variable to store the name of a student. If a student has a long name of say 20 characters, then the string can be expanded to accommodate 20 characters. On the other hand, a student name has only 5 characters, then the string variable can be contracted to store only 5 characters. However, to use a variable-length string format you need a technique to indicate the end of elements that are a part of the string. This can be done either by using length-controlled string or a delimiter.
- **Length-controlled string** In a length-controlled string, you need to specify the number of characters in the string. This count is used by string manipulation functions to determine the actual length of the string variable.
- **Delimited string** In this format, the string is ended with a delimiter. The delimiter is then used to identify the end of the string. For example, in English language every sentence is ended with a full-stop (.). Similarly, in C we can use any character such as comma, semicolon, colon, dash, null character, etc. as the delimiter of a string. However, null character is the most commonly used string delimiter in the C language

Common Operations:

- strcpy(): Copy a string.
- strcat(): Concatenate two strings.
- strlen(): Find the length of a string.
- strcmp(): Compare two strings.

Q4 (b) Explain various operations on strings.

1. **strlen()** - *Find Length of a String*

Syntax: size_t strlen(const char *str);
Description: Returns the length of the string (excluding the null character \0).

Example:
#include <stdio.h>

```
#include <string.h>

int main() {
   char str[] = "Hello";
   printf("Length of string: %lu\n", strlen(str));
   return 0;
}
```

Output: Length of string: 5

2. **strcpy()** - *Copy One String to Another*

Syntax: char *strcpy(char *dest, const char *src);

Description: Copies the contents of src into dest (including the null character).

Example:

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

int main() {
   char source[] = "CMRIT";
   char destination[20];
   strcpy(destination, source);
   printf("Copied String: %s\n", destination);
   return 0;
     }
```

Output: Copied String: CMRIT

3. **strcmp()** - *Compare Two Strings*

Syntax: int strcmp(const char *str1, const char *str2);

Description: Compares two strings and returns:

0 if both are equal
A negative value if str1 < str2
A positive value if str1 > str2

Example:

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

int main() {
   char str1[] = "Hello";
   char str2[] = "World";

   if (strcmp(str1, str2) == 0)
```

```
        printf("Strings are equal.\n");
    else
        printf("Strings are not equal.\n");

    return 0;
}
```

Output: Strings are not equal.

4. **strcat()** - *Concatenate Two Strings*

Syntax: char *strcat(char *dest, const char *src);

Description: Appends src to dest, modifying dest.

Example:

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

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("Concatenated String: %s\n", str1);
    return 0;
}
```

Output: Concatenated String: Hello, World!

5. **strrev()** - *Reverse a String (Non-Standard)*

Syntax: char *strrev(char *str);

Description: Reverses the given string in place (not a standard function in <string.h>, but available in some compilers).

Example:

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

int main() {
    char str[] = "CMRIT";
    printf("Reversed String: %s\n", strrev(str));
    return 0;
}
```

Output: Reversed String: TIRMC

Q4 (c) Write a program to search an element using linear search.

```
#include <stdio.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;  // Return index if target is found
        }
    }
    return -1;  // Return -1 if target is not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int target = 30;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = linearSearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Module 3

Q5 (a) Define function. Differentiate between Call by value and call by reference.

Answer:
A **function** in C is a block of code that performs a specific task. It is used to avoid redundancy, improve modularity, and make code reusable. Functions can take parameters and return a value.

In **Call by Value**, the actual value of the argument is passed to the function. This means that any changes made to the parameter inside the function do not affect the actual argument used in the function call.

- The function gets a copy of the value of the argument.
- Changes to the parameter in the function do not affect the original variable.

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

```
void modifyValue(int num) {
   num = 100;  // Modifying the local copy
}

int main() {
   int a = 10;
   modifyValue(a);  // Call by value
   printf("Value of a after function call: %d\n", a);  // Output will be 10
   return 0;
}
```

In **Call by Reference**, the address (reference) of the argument is passed to the function, allowing the function to modify the actual argument's value directly.

- The function receives the memory address of the argument, so any changes made to the parameter inside the function directly affect the original argument.

```
#include <stdio.h>

void modifyValue(int *num) {
   *num = 100;  // Modifying the value at the address pointed by num
}

int main() {
   int a = 10;
   modifyValue(&a);  // Call by reference
   printf("Value of a after function call: %d\n", a);  // Output will be 100
   return 0;
}
```

Q5 (b) Recursive function to find Fibonacci series.

```
#include<stdio.h>

int fib(int n) {
   if(n<=1)
      return n;
   return fib(n-1)+fib(n-2);
}

int main() {
   int n,i;
   scanf("%d",&n);

   for(i=0;i<n;i++)
      printf("%d ",fib(i));
}
```

4-c) Swap without temp

```
#include<stdio.h>

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

   a=a+b;
   b=a-b;
   a=a-b;

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

Q6 (a) Write a C program to find mean of N numbers using array and pointers.

```
#include <stdio.h>

int main() {
   int n, i;
   float sum = 0.0, mean;

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

   float numbers[n];  // Array to store numbers
   float *ptr = numbers;  // Pointer to the array

   // Input elements
   printf("Enter %d numbers:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%f", ptr + i);  // Using pointer arithmetic
   }

   // Calculate sum using pointers
   for (i = 0; i < n; i++) {
      sum += *(ptr + i);  // Dereferencing pointer
   }

   mean = sum / n;
   printf("Mean = %.2f\n", mean);

   return 0;
}
```

Q6 (b) Write a C rogram to add two matrices using pointers.

```
#include <stdio.h>

int main() {
```

```
    int m, n, i, j;

    printf("Enter number of rows and columns: ");
    scanf("%d%d", &m, &n);

    int a[m][n], b[m][n], sum[m][n];

    // Input first matrix
    printf("Enter elements of Matrix A:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", (*(a + i) + j));
        }
    }

    // Input second matrix
    printf("Enter elements of Matrix B:\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", (*(b + i) + j));
        }
    }

    // Add matrices using pointers
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            *(*(sum + i) + j) = *(*(a + i) + j) + *(*(b + i) + j);
        }
    }

    // Display result
    printf("Resultant Matrix (A + B):\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", *(*(sum + i) + j));
        }
        printf("\n");
    }

    return 0;
}
```

Module 4

Q7 (a) Define structure. Give syntax of structure.

A **structure** in C is a user-defined data type that allows grouping variables of **different types** under one name. It is used to represent a **record** (like a student, employee, etc.).

Syntax of Structure:

```
struct StructureName {
   data_type member1;
   data_type member2;
   // ...
};
```

Example:

```
struct Student {
   int rollNo;
   char name[50];
   float marks;
};
```

Q7 (b) Write a C program using structure to read and display student information.

```
#include <stdio.h>

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

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

   // Read student details
   printf("Enter student roll number: ");
   scanf("%d", &s.rollNo);

   printf("Enter student name: ");
   scanf(" %[^\n]", s.name);  // To read string with spaces

   printf("Enter student marks: ");
   scanf("%f", &s.marks);

   // Display student details
   printf("\n--- Student Information ---\n");
   printf("Roll Number: %d\n", s.rollNo);
   printf("Name       : %s\n", s.name);
   printf("Marks      : %.2f\n", s.marks);

   return 0;
}
```

Q7 (c) Explain Nested structure with example.

A **nested structure** in C is a structure that contains another structure as one of its members. This helps in organizing complex data logically — like grouping address inside a student or employee record.

 Syntax of Nested Structure:

c
CopyEdit
```
struct Inner {
   // members
};

struct Outer {
   struct Inner innerMember;  // Nested structure
   // other members
};
```

Example: Student with Address (Nested Structure)

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

// Inner structure
struct Address {
   char city[30];
   int pin;
};

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

int main() {
   struct Student s;

   // Input
   printf("Enter roll number: ");
   scanf("%d", &s.rollNo);

   printf("Enter name: ");
   scanf(" %[^\n]", s.name);
```

```
    printf("Enter city: ");
    scanf(" %[^\n]", s.addr.city);

    printf("Enter pin code: ");
    scanf("%d", &s.addr.pin);

    // Output
    printf("\n--- Student Details ---\n");
    printf("Roll No : %d\n", s.rollNo);
    printf("Name    : %s\n", s.name);
    printf("City    : %s\n", s.addr.city);
    printf("PIN     : %d\n", s.addr.pin);

    return 0;
}
```

Q8 (a) Define union. Explain the syntax of Union with example.

A union in C is a user-defined data type similar to a structure, but with a key difference:

In a union, all members share the same memory location. This means only one member can contain a value at any given time, saving memory when variables are not used simultaneously.

Syntax of Union:

```
union UnionName {
    data_type member1;
    data_type member2;
    // ...
};
```

Example:

```
union Data {
    int i;
    float f;
    char str[20];
};
```

Declaring a Union Variable:

```
union Data d1;
```

Accessing Union Members:

```
d1.i = 10;
```

d1.f = 3.14;

Q8 (b) Explain the various storage classes.

Storage classes in C define scope, lifetime, default value, and visibility of variables. There are four types:

1. auto Storage Class

The auto storage class is the default for all local variables inside a function. It is automatically applied to any local variable that is declared inside a function without explicitly specifying a storage class. The scope of an auto variable is local to the block in which it is defined, meaning it can only be accessed within that block. The lifetime of the variable ends when the block or function ends, and it is created when the block is entered and destroyed when it exits. By default, if not initialized, auto variables contain garbage values.

Syntax:

auto data_type variable_name;

Example:

```
#include <stdio.h>

int main() {
   auto int x = 5;  // 'auto' is implicit, so you could just write 'int x = 5;'
   printf("Value of x: %d\n", x);
   return 0;
}
```

Explanation: In this example, x is an auto variable (even though the auto keyword is optional). Its value is printed within the function, and the variable is destroyed once the function ends.

---

2. register Storage Class

The register storage class suggests to the compiler that a variable should be stored in a CPU register for faster access instead of RAM. This storage class is typically used for variables that are frequently accessed, like loop counters. However, it's important to note that you cannot take the address of a register variable using the address-of operator (&), because registers may not be stored in regular memory.

Syntax:

register data_type variable_name;

Example:

```
#include <stdio.h>

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

---

3. static Storage Class

The static storage class is used to retain the value of a variable between function calls. If a variable is declared as static within a function, it retains its value between calls to that function. It is initialized only once, and its lifetime is the entire duration of the program. If declared globally, static restricts the scope of the variable to the file in which it is declared (i.e., it is not visible to other files).

Syntax:

```
c
CopyEdit
static data_type variable_name;
```

Example:

```
#include <stdio.h>

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

int main() {
   count_calls();  // Output: Call count: 1
   count_calls();  // Output: Call count: 2
   count_calls();  // Output: Call count: 3
   return 0;
}
```

---

4. extern Storage Class

The extern storage class is used to declare a variable that is defined in another file or elsewhere in the same file. It tells the compiler that the variable exists but does not allocate

space for it. The variable should be defined in another part of the program, typically in a different file. extern is used to share variables across multiple files in large projects.

Syntax:

extern data_type variable_name;

Example:

```
// File1.c
#include <stdio.h>

extern int x;  // Declare the external variable

int main() {
    printf("Value of x: %d\n", x);
    return 0;
}
// File2.c
#include <stdio.h>

int x = 10;  // Definition of the external variable
```

Q8 (c) Write a short note on typedef.

The typedef keyword in C is used to create new type names (aliases) for existing data types. It helps improve code readability, especially when dealing with complex data types like pointers, structures, and arrays. By using typedef, you can give more meaningful names to these types, making the code more understandable and easier to manage.

Syntax:

```
typedef existing_type new_type_name;
Example:
#include <stdio.h>

typedef int Integer;  // 'Integer' is now an alias for 'int'

int main() {
    Integer x = 10;  // Using 'Integer' instead of 'int'
    printf("Value of x: %d\n", x);
    return 0;
}
```

Module 5

Q9. a. Define a File. List and explain the operations on file.

- Definition:
  A file in C is a storage medium for saving data permanently, accessible through pointers using functions from stdio.h.
- Common Operations:
  1. Opening a file – fopen("filename", "mode");
     Example: FILE *fp = fopen("data.txt", "r");
  2. Reading/Writing data – fscanf(), fgets(), fprintf(), etc.
     Example: fscanf(fp, "%d", &num);
  3. File positioning – fseek(), ftell(), etc.
     Example: fseek(fp, 0, SEEK_END);
  4. Closing a file – fclose(fp);
     Example: fclose(fp);

---

9. b. List and describe the file modes.

Answer:

File modes in C specify the **purpose for which a file is opened** using the `fopen()` function. They determine whether the file is opened for **reading, writing, or appending**.

Syntax of fopen()

```
FILE *fp;
fp = fopen("filename", "mode");
```

Types of File Modes

Mode	Meaning	Description
`"r"`	Read	Opens file for reading; file must exist
`"w"`	Write	Creates new file or overwrites existing file
`"a"`	Append	Adds data at the end of file
`"r+"`	Read & Write	Opens existing file for both operations
`"w+"`	Write & Read	Creates new file; allows read & write
`"a+"`	Append & Read	Opens file for reading and appending

Explanation of Modes

1. Read Mode ("r")

- File must exist
- Pointer at beginning

```
fp = fopen("data.txt", "r");
```

2. Write Mode ("w")

- Creates new file
- If file exists → old data deleted

```
fp = fopen("data.txt", "w");
```

3. Append Mode ("a")

- Adds data at end
- Existing data preserved

```
fp = fopen("data.txt", "a");
```

4. Read & Write Mode ("r+")

- File must exist
- Allows both reading and writing

5. Write & Read Mode ("w+")

- Creates new file
- Allows both operations

6. Append & Read Mode ("a+")

- Opens file for reading and appending

Q10. a.

Answer:

Functions for Reading Strings

1. `scanf()`
2. `gets()`
3. `fgets()`

---

1. scanf() Function

`scanf()` is used to read formatted input from the user.

Syntax:

```
scanf("%s", str);
```

- Reads input **until space, tab, or newline**
- Cannot read full sentence with spaces

Example:

```
#include<stdio.h>
int                                  main()                                  {
    char                                                                 name[20];
    scanf("%s",                                                             name);
    printf("%s",                                                            name);
}
```

Limitation: Stops reading at first space

---

2. gets() Function

`gets()` reads a **complete string including spaces** from standard input.

Syntax:

```
gets(str);
```

Example:

```
#include<stdio.h>
int                                  main()                                  {
    char                                                                 name[50];
    gets(name);
    printf("%s",                                                            name);
}
```

3. fgets() Function

`fgets()` reads a string from input **including spaces** and is safer than `gets()`.

Syntax:

```
fgets(str, size, stdin);
```

- Reads at most `size-1` characters
- Includes newline character (`\n`)

Example:

```
#include<stdio.h>
int                                  main()                                  {
```

```
    char                                                    name[50];
    fgets(name,                         50,                   stdin);
    printf("%s",                                               name);
}
```

Q10. b.

In C programming, strings can be displayed or written using various **output functions**. These functions help in printing strings to the **standard output (screen)** or writing them to files.

Functions for Writing Strings

1. `printf()`
2. `puts()`
3. `fputs()`

---

1. printf() Function

Definition:

`printf()` is used to print formatted output to the screen.

Syntax:

```
printf("%s", str);
```

- Prints string until **null character (`'\0'`)**
- Allows formatting (like `%d`, `%f`, `%s`)

Example:

```
#include<stdio.h>
int                                 main()                                  {
    char                  name[]                  =                  "Pooja";
    printf("%s",                                                       name);
}
```

2. puts() Function

`puts()` is used to display a string on the screen.

Syntax:

```
puts(str);
```

- Prints string followed by **newline automatically**

Example:

```
#include<stdio.h>
int                                  main()                                  {
    char                  name[]                  =                  "Pooja";
    puts(name);
}
```

3. fputs() Function

`fputs()` is used to write a string to a file or output stream.

Syntax:

```
fputs(str, stdout);
```

Example:

```
#include<stdio.h>
int                                  main()                                  {
    char                  name[]                  =                  "Pooja";
    fputs(name,                                                           stdout);
}
```