Module 2 – Introduction to Programming

Q1. Overview of C Programming:

THEORY EXERCISE:

Write an essay covering the history and evolution of C programming. Explain

its importance and why it is still used today.

Ans:

The History and Evolution of C Programming

C was developed by **Dennis Ritchie** at **Bell Labs** between **1969 and 1973**, derived from the **B language** and designed to rewrite **UNIX**. It became widely used due to its **efficiency, portability, and flexibility**.

*Evolution of C:*

1. **K&R C (1978)** – First standard described in *The C Programming Language* by Kernighan & Ritchie.
2. **ANSI C (1989)** – Standardized by ANSI (American National Standards Institute) to ensure portability.
3. **C99 (1999)** – Introduced inline functions, new data types, and variable-length arrays.
4. **C11 (2011) & C18 (2018)** – Added multi-threading support, improved security, and minor refinements.

*Importance & Relevance Today*

C remains popular due to its **speed, control over hardware, and efficiency**. It is widely used in **operating systems (Linux, Windows), embedded systems, game development, and compilers**. Its influence extends to modern languages like **C++, Java, and Python**, ensuring its continued relevance in programming.

LAB EXERCISE:

Research and provide three real-world applications where C programming is

extensively used, such as in embedded systems, operating systems, or game

Development.

Ans:

**Real-World Applications of C Programming**

1. **Operating Systems** – C is the backbone of OS development.

**Examples:** Linux Kernel, Windows, macOS.

1. **Embedded Systems** – Used in hardware-level programming.

**Examples:** Automotive ECUs, medical devices, smart appliances.

1. **Game Development** – Ensures high performance in gaming engines.

**Examples:** Unreal Engine, Doom (1993), console games.

C remains essential for **efficiency, hardware control, and speed**, making it crucial in various industries.

Q2. Setting Up Environment

THEORY EXERCISE:

Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated

Development Environment (IDE) like DevC++, VS Code, or CodeBlocks.

**Steps to Install a C Compiler and Set Up an IDE**

***1. Installing GCC (GNU Compiler Collection)***

GCC is a widely used C compiler available for multiple platforms.

**For Windows:**

* Download **MinGW** (Minimalist GNU for Windows) from <https://osdn.net/projects/mingw/>.
* Run the installer and select **"mingw-gcc"** from the package list.
* Add the **bin** folder path (e.g., C:\MinGW\bin) to the system **Environment Variables** for command-line access.
* Verify installation using gcc --version in the command prompt.**2. Installing and Setting Up DevC++ (Windows Only)**

DevC++ is a lightweight IDE with a built-in compiler.

* Download DevC++ from <https://sourceforge.net/projects/orwelldevcpp/>.
* Run the installer and follow on-screen instructions.
* Open DevC++ and configure the compiler:
* Go to **Tools > Compiler Options** and select **TDM-GCC** as the compiler.
* Create a new C project, write a sample program, and compile using **F11** (Run&Compile).

**LAB EXERCISE:**

Install a C compiler on your system and configure the IDE. Write your first

program to print "Hello, World!" and run it.

Ans:

#include <stdio.h>

 main() {

    printf("Hello, World!\n");

}

Q3. Basic Structure of a C Program

THEORY EXERCISE:

Explain the basic structure of a C program, including headers, main function,

comments, data types, and variables. Provide examples.

Ans:

**Basic Structure of a C Program**

A C program follows a specific structure, consisting of **headers, the main function, comments, data types, and variables**. Below is a breakdown of each component with examples.

**1. Header Files**

Header files contain standard functions and libraries. The #include directive is used to import them.

**Example:**

#include <stdio.h>  // Includes the standard input-output library   
# - it’s pre-defined , include – add something , stdio – Standard input-output , (.h) - header file

**2. Main Function (main())**

Every C program starts execution from the main() function.

**Example:**

 main() {   
     
} 

**3. Comments**

Comments help document the code and are ignored by the compiler.

* **Single-line comment:** // This is a comment
* **Multi-line comment:**

/\*    
   This is a multi-line comment    
\*/ 

**4. Data Types and Variables**

Data types define the type of data a variable can store.

**Common Data Types in C:**

* int – Stores integers (e.g., 10, -5). [%d]
* float – Stores decimal numbers (e.g., 3.14).[%f]
* char – Stores a single character (e.g., 'A').[%c]
* double – Stores large floating-point numbers.(124555.3625)[%lf]
* Long double-Store long value(5552211448866332)[%Lf]
* Boolean – Store true, false value (true, false) [1,0]

**Example:**

int age = 25;         // Integer variable   
float price = 10.5;   // Floating-point variable   
char grade = 'A';     // Character variable 

**5. Complete Example Program**

#include <stdio.h>  // Header file   
   
main() {     
    // Variable declaration ,data-member    
    int number = 10;     
    float pi = 3.14;     
    char letter = 'C';     
       
    // Printing values   
    printf("Number: %d\n", number);     
    printf("Value of Pi: %.2f\n", pi);     
    printf("Character: %c\n", letter);     
       
      
} 

**Output:**

Number: 10     
Value of Pi: 3.14     
Character: C   

LAB EXERCISE:

Write a C program that includes variables, constants, and comments. Declare and use different data types (int, char, float) and display their values.

Ans:

 main() {

    int age = 25;

    float price = 99.99;

    char grade = 'A';

    printf("Age: %d\n", age);

    printf("Price: %.2f\n", price);

    printf("Grade: %c\n", grade);

}

Output:

Age: 25

Price: 99.99

Grade: A

Q5. Control Flow Statements in C

THEORY EXERCISE:

Explain decision-making statements in C (if, else, nested if-else, switch).

Provide examples of each.

**if-else Statement**

Executes one block if the condition is **true**, otherwise executes another block.

**Example:**

#include <stdio.h>

 main() {

    int num = 10;

    if (num % 2 == 0) {

        printf("Even number\n");

    } else {

        printf("Odd number\n");

    }

}

Output : Even number

**Nested if-else Statement**

An if statement inside another if or else block.

**Example:**

#include <stdio.h>

 main() {

    int num = 10;

    if (num > 0) {

        if (num % 2 == 0) {

            printf("Positive even number\n");

        } else {

            printf("Positive odd number\n");

        }

    } else {

        printf("Non-positive number\n");

    }

}

Output : Positive even number

**switch Statement**

Used when there are multiple conditions to check. It evaluates an expression and executes the matching case.

**Syntax:**

switch(expression) {

    case value1:

        // Code to execute

        break;

    case value2:

        // Code to execute

        break;

    default:

        // Code if no case matches

}

Example:

#include <stdio.h>

 main() {

    int choice = 2;

    switch (choice) {

        case 1:

            printf("You chose option 1\n");

            break;

        case 2:

            printf("You chose option 2\n");

            break;

        default:

            printf("Invalid choice\n");

    }

}

Output : You chose option 2

 Write a C program to check if a number is even or odd using an if-else statement. Extend the program using a switch statement to display the monthname based on the user’s input (1 for January, 2 for February, etc.)

int main() {

int num, month;

printf("Enter a number to check if it is even or odd: ");

scanf("%d", &num);

if (num % 2 == 0) {

printf("The number %d is even.\n", num);

} else {

printf("The number %d is odd.\n", num);

}

printf("Enter a number between 1 and 12 to get the corresponding month name: ");

scanf("%d", &month);

switch(month) {

case 1:

printf("Month: January\n");

break;

case 2:

printf("Month: February\n");

break;

case 3:

printf("Month: March\n");

break;

case 4:

printf("Month: April\n");

break;

case 5:

printf("Month: May\n");

break;

case 6:

printf("Month: June\n");

break;

case 7:

printf("Month: July\n");

break;

case 8:

printf("Month: August\n");

break;

case 9:

printf("Month: September\n");

break;

case 10:

printf("Month: October\n");

break;

case 11:

printf("Month: November\n");

break;

case 12:

printf("Month: December\n");

break;

default:

printf("Invalid input! Please enter a number between 1 and 12.\n");

}

}

Q6.Looping in C

THEORY EXERCISE:

Compare and contrast while loops, for loops, and do-while loops. Explain the

scenarios in which each loop is most appropriate.

**Comparison of Loops in C: while, for, and do-while**

C provides three types of loops: **while, for, and do-while**, each suited for different scenarios.

**1. while Loop**

* **Structure:** while (condition) {   
      // Code block   
  }
* **How it Works:**
* Checks the **condition first**.
* Executes **only if** the condition is **true**.
* If the condition is **false initially**, the loop **won’t run** at all.
* **Best Used When:**
* The number of iterations is **unknown** beforehand.
* Loop runs based on **user input or sensor data**.

**Example:** Reading input until the user enters 0

#include <stdio.h>   
main() {   
    int num = 1;   
    while (num != 0) {   
        printf("Enter a number (0 to stop): ");   
        scanf("%d", &num);   
    }   
     
} 

**2. for Loop**

* **Structure:** for (initialization; condition; update) {   
      // Code block   
  }
* **How it Works:**
* Initializes a variable.
* Checks the **condition** before each iteration.
* Updates the variable **after each loop cycle**.
* **Best Used When:**
* The number of iterations is **known** beforehand.
* Looping over **arrays, counters, or fixed ranges**.

**Example:** Printing numbers from 1 to 10

#include <stdio.h>   
main() {   
    for (int i = 1; i <= 10; i++) {   
        printf("%d\n", i);   
    }   
       
} 

**3. do-while Loop**

* **Structure:** do {   
      // Code block   
  } while (condition);
* **How it Works:**
* Executes the **code at least once**, then checks the condition.
* If the condition is true, it repeats.
* **Best Used When:**
* The loop **must execute at least once**.
* Used in **menus, user interaction, and validation**.

**Example:** int main() {

int i = 1;

do {

printf("%d\n", i);

i++;

} while (i <= 10);

}

Q.7 Loop Control Statements

THEORY EXERCISE:

 Explain the use of break, continue, and goto statements in C. Provide

examples of each.

**Control Statements in C: break, continue, and goto**

In C programming, **control statements** are used to change the normal flow of execution. Three important control statements are:

1. **break Statement**
2. **continue Statement**
3. **goto Statement**

Each of these plays a crucial role in controlling loops and other structures in C.

**1. break Statement**

The break statement is used to **immediately terminate a loop or switch case** when a specific condition is met. After encountering break, the control moves to the statement following the loop or switch case.

**Uses of break**

* Exiting a for, while, or do-while loop prematurely.
* Terminating a switch statement once a case is matched.

**Example Use Case**

In a loop, break is often used to stop execution when a condition is met.

for (int i = 1; i <= 5; i++) {   
    if (i == 3) {   
        break;  // Loop terminates when i == 3   
    }   
    printf("%d ", i);   
} 

**Output:**

1 2 

 Here, when i == 3, the break statement stops the loop.

**2. continue Statement**

The continue statement **skips the current iteration of the loop and moves to the next iteration**. Unlike break, it does not terminate the loop but rather forces the next cycle of execution.

**Uses of continue**

* Skipping specific iterations in a loop based on a condition.
* Avoiding unnecessary computations inside a loop.

**Example Use Case**

If we want to skip a specific value in a loop, continue can be used.

for (int i = 1; i <= 5; i++) {   
    if (i == 3) {   
        continue;  // Skips printing when i == 3   
    }   
    printf("%d ", i);   
} 

**Output:**

1 2 4 5 

Here, when i == 3, the continue statement skips the printf statement and moves to the next iteration.

**3. goto Statement**

The goto statement allows for an **unconditional jump** to another part of the program using a label. While it provides flexibility, excessive use can lead to **unstructured and hard-to-read code**. It is generally discouraged in modern programming.

**Uses of goto**

* Exiting deeply nested loops.
* Jumping forward or backward in code execution.

**Example Use Case**

A simple example of goto for repeating a block of code:

#include <stdio.h>   
main() {   
    int num = 1;   
   
start:   
    printf("%d ", num);   
    num++;   
       
    if (num <= 5) {   
        goto start;      }   
      
} 

**Output:** 1 2 3 4 5

Write a C program that uses the break statement to stop printing numbers when it reaches 5. Modify the program to skip printing the number 3 usingthecontinue statement.

int main() {

for (int i = 1; i <= 10; i++) {

if (i == 5) {

break; // Stop the loop when i reaches 5

}

if (i == 3) {

continue; // Skip printing 3

}

printf("%d\n", i); // Print the current number

}

}

Q.8 . Functions in C

THEORY EXERCISE:

What are functions in C? Explain function declaration, definition, and how to

call a function. Provide examples.

**Functions in C (Simple Explanation)**

A **function** in C is a block of code that performs a specific task. It helps in making the program **modular**, **reusable**, and **easy to read**.

**Parts of a Function**

A function has **three main parts**:

1. **Function Declaration** – Declares the function before using it.
2. **Function Definition** – Contains the actual code of the function.
3. **Function Call** – Executes the function.

**1. Function Declaration (Prototype)**

Tells the compiler about the function **before using it**.

**Syntax:**

return\_type function\_name(parameters); 

**Example:**

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

* int → Function returns an integer.
* add → Function name.
* (int, int) → Two integer parameters.

**2. Function Definition**

Defines what the function does.

**Syntax:**

return\_type function\_name(parameters) {   
    // Function code   
    return value;  // (if needed)   
} 

**Example:**

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

**3. Function Call**

Executes the function from main().

**Example:**

int sum = add(5, 10);  // Calling the function 

**Complete Example:**

#include <stdio.h>   
   
   
int add()

{

int num1 = 5, num2 = 10; 

  int result = num1+num2;   
       
    printf("Sum: %d\n", result);

}   
   
int main() { 

add();    
} 

**Output:**

Sum: 15

Write a C program that calculates the factorial of a number using a function. Include function declaration, definition, and call.

int factorial(int n);

int main() {

int number, result;

printf("Enter a number: ");

scanf("%d", &number);

result = factorial(number);

printf("Factorial of %d is %d\n", number, result);

}

int factorial(int n) {

int fact = 1;

for (int i = 1; i <= n; i++) {

fact \*= i;

}

}

Q.9 Arrays in C

THEORY EXERCISE:

Explain the concept of arrays in C. Differentiate between one-dimensional and

multi-dimensional arrays with examples.

**Arrays in C**

**What is an Array?**

An **array** in C is a collection of elements of the **same data type** stored in **contiguous memory locations**. Arrays allow efficient access and manipulation of multiple values using a **single variable name**.

**Key Features of Arrays:**

* Stores multiple values of the **same type**.
* Uses **indexing** (starting from 0) to access elements.
* Reduces code complexity by avoiding multiple individual variables.

**Types of Arrays**

1️⃣ **One-Dimensional (1D) Array**

A **1D array** stores elements in a single row (linear structure).

2️⃣ **Multi-Dimensional Array**

An array with more than one dimension, such as:

* **Two-Dimensional (2D) Array** → Stores data in rows and columns (like a table).
* **Three-Dimensional (3D) Array** → Stores multiple tables of data.

**1. One-Dimensional Array (1D Array)**

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

**Declaration & Initialization:**

data\_type array\_name[size]; 

**Example:**

#include <stdio.h>   
   
int main() {   
    int numbers[5] = {10, 20, 30, 40, 50};  // 1D Array   
       
    printf("First element: %d\n", numbers[0]);  // Accessing first element   
    printf("Second element: %d\n", numbers[1]);   
   
    return 0;   
} 

**Memory Representation:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Index** | **0** | **1** | **2** | **3** | **4** |
| Values | 10 | 20 | 30 | 40 | 50 |

**2. Multi-Dimensional Arrays**

**A) Two-Dimensional Array (2D Array)**

A **2D array** stores data in **rows and columns** (like a table).

**Declaration & Initialization:**

data\_type array\_name[rows][columns]; 

**Example:**

#include <stdio.h>   
   
int main() {   
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };  // 2D Array   
       
    printf("Element at row 1, col 2: %d\n", matrix[0][1]);  // Accessing element (row 1, col 2)   
   
    return 0;   
} 

**Memory Representation:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Row \ Col** | **0** | **1** | **2** |
| **0** | 1 | 2 | 3 |
| **1** | 4 | 5 | 6 |

**B) Three-Dimensional Array (3D Array)**

A **3D array** stores multiple **2D tables**.

**Declaration:**

data\_type array\_name[x][y][z]; 

**Example:**

int cube[2][2][2] = {   
    {{1, 2}, {3, 4}},   
    {{5, 6}, {7, 8}}   
}; 

It represents **two** 2D arrays (like pages of a book).

**Differences Between 1D and Multi-Dimensional Arrays**

|  |  |  |
| --- | --- | --- |
| **Feature** | **One-Dimensional Array (1D)** | **Multi-Dimensional Array (2D/3D)** |
| Structure | Linear (single row) | Table-like (rows & columns) |
| Indexing | Uses **one** index [i] | Uses **multiple** indexes [i][j], [i][j][k] |
| Memory Use | Less | More (stores complex data) |
| Example | {10, 20, 30} | {{1,2}, {3,4}} (2D) |

Write a C program that stores 5 integers in a one-dimensional array andprintsthem. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.

int main() {

int arr1D[5] = {1, 2, 3, 4, 5};

printf("One-dimensional array :\n");

for (int i = 0; i < 5; i++) {

printf("%d ", arr1D[i]);

}

printf("\n");

int arr2D[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

printf("Two-dimensional array (3x3 matrix):\n");

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

printf("%d ", arr2D[i][j]);

}

printf("\n");

}

int sum = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

sum += arr2D[i][j];

}

}

printf("\nSum of all elements in the 2D array: %d\n", sum);

}

Q.10 Pointers in C

THEORY EXERCISE:

 Explain what pointers are in C and how they are declared and initialized. Why

are pointers important in C?

**Pointers in C**

**What is a Pointer?**

A **pointer** in C is a variable that **stores the memory address** of another variable. Instead of storing an actual value, a pointer holds the **location** where the value is stored in memory.

**Key Features of Pointers:**

* Store memory addresses instead of actual values.
* Allow **direct memory access and manipulation**.
* Help in **dynamic memory allocation** and efficient data handling.

**Declaring and Initializing a Pointer**

**Syntax:**

data\_type \*pointer\_name; 

* \* (asterisk) indicates that the variable is a pointer.
* data\_type is the type of data the pointer will point to.

**Why are Pointers Important in C?**

1. **Efficient Memory Management:**
2. Used in **dynamic memory allocation** (e.g., malloc, free).
3. **Passing by Reference:**
4. Allows functions to modify actual variables instead of making copies.
5. **Array and String Handling:**
6. Pointers help in **iterating arrays and handling strings efficiently**.
7. **Improves Performance:**
8. Direct memory access is **faster than accessing through variable names**.
9. **Data Structures Implementation:**
10. Used in **linked lists, trees, stacks, and queues**.

 Write a C program to demonstrate pointer usage. Use a pointer to modifythevalue of a variable and print the result.

int main() {

int num = 10;

int \*ptr;

ptr = &num;

printf("Before modification:\n");

printf("Value of num: %d\n", num);

printf("Value at pointer ptr: %d\n", \*ptr);

\*ptr = 20;

printf("\nAfter modification:\n");

printf("Value of num: %d\n", num);

printf("Value at pointer ptr: %d\n", \*ptr);

}

Q.11 Strings in C

THEORY EXERCISE:

Explain string handling functions like strlen(), strcpy(), strcat(),

strcmp(), and strchr(). Provide examples of when these functions are

Useful

**String Handling Functions in C**

Strings in C are **character arrays** ending with a **null character (\0)**. The <string.h> library provides various functions to handle strings efficiently.

**1. strlen() – String Length**

* **Purpose:** Returns the length of a string (excluding \0).
* **Syntax:**

size\_t strlen(const char \*str); 

* **Example:**

#include <stdio.h> 

 main() {   
    char str[] = "Hello";   
    printf("Length of string: %lu\n", strlen(str)); // Output: 5   
       
} 

* **Use Case:** To determine the length of a string for loops or memory allocation.

**2. strcpy() – String Copy**

* **Purpose:** Copies one string into another.
* **Syntax:**

char \*strcpy(char \*dest, const char \*src); 

* **Example:**

#include <stdio.h>   
main() {   
    char source[] = "C Programming";   
    char destination[20];   
   
    strcpy(destination, source);   
    printf("Copied String: %s\n", destination);   
   
      
} 

* **Use Case:** To copy strings safely between variables.

**3. strcat() – String Concatenation**

* **Purpose:** Appends one string to another.
* **Syntax:**

char \*strcat(char \*dest, const char \*src); 

* **Example:**

#include <stdio.h>   
   
main() {   
    char str1[20] = "Hello, ";   
    char str2[] = "World!";   
   
    strcat(str1, str2);   
    printf("Concatenated String: %s\n", str1);   
   
       
} 

* **Use Case:** Used when combining strings dynamically.

**4. strcmp() – String Comparison**

* **Purpose:** Compares two strings lexicographically.
* **Syntax:**

int strcmp(const char \*str1, const char \*str2); 

* Returns **0** if strings are equal.
* Returns **< 0** if str1 is smaller.
* Returns **> 0** if str1 is greater.
* **Example:**

#include <stdio.h>   
main() {   
    char str1[] = "apple";   
    char str2[] = "banana";   
   
    if (strcmp(str1, str2) == 0)   
        printf("Strings are equal\n");   
    else   
        printf("Strings are different\n");   
   
       
} 

* **Use Case:** Useful for **sorting**, **searching**, and **password verification**.

**5. strchr() – Find a Character in a String**

* **Purpose:** Searches for a character in a string.
* **Syntax:**

char \*strchr(const char \*str, int ch); 

* Returns a **pointer** to the first occurrence of ch in str, or NULL if not found.
* **Example:**

#include <stdio.h>   
main() {   
    char str[] = "Hello, World!";   
    char \*ptr = strchr(str, 'W');   
   
    if (ptr)   
        printf("Character found at position: %ld\n", ptr - str);   
    else   
        printf("Character not found\n");   
   
      
} 

* **Use Case:** Useful for **parsing text** and **searching for characters** in user inputs.

 Write a C program that takes two strings from the user and concatenates themusing strcat(). Display the concatenated string and its length using strlen().

#include <stdio.h>

#include <string.h>

int main() {

char str1[100], str2[100];

printf("Enter the first string: ");

fgets(str1, sizeof(str1), stdin);

str1[strcspn(str1, "\n")] = '\0';

printf("Enter the second string: ");

fgets(str2, sizeof(str2), stdin);

str2[strcspn(str2, "\n")] = '\0';

strcat(str1, str2);

printf("\nConcatenated string: %s\n", str1);

printf("Length of concatenated string: %zu\n", strlen(str1));

}

Q.12 Structures in C

**Structures in C**

**What is a Structure?**

A **structure** in C is a user-defined data type that allows **grouping of different data types** under a single name. It is useful for storing **related information** together.

**Declaring a Structure**

A structure is declared using the struct keyword.

**Syntax:**

struct structure\_name {   
    data\_type member1;   
    data\_type member2;   
    ...   
}; 

**Example:**

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

**1. Declaring and Initializing a Structure Variable**

struct Student s1 = {101, "Kartik", 95.5}; 

**2. Accessing Structure Members using . Operator**

#include <stdio.h>   
   
struct Student {   
    int id;   
    char name[50];   
    float marks;   
};   
   
main() {   
    struct Student s1 = {101, "Kartik", 95.5};  // Initialize structure   
   
    printf("Student ID: %d\n", s1.id);   
    printf("Student Name: %s\n", s1.name);   
    printf("Marks: %.2f\n", s1.marks);   
   
       
} 

**Output:**

Student ID: 101     
Student Name: Kartik     
Marks: 95.50   

**Why Use Structures?**

* Group **different data types** together.
* Improves **code organization**.
* Useful for **complex data management** (e.g., databases, records).
* Makes **passing multiple values to functions** easier.

Write a C program that defines a structure to store a student's details (name, roll number, and marks). Use an array of structures to store details of 3students and print them.

struct Student {

char name[50];

int rollNumber;

float marks;

};

int main() {

struct Student students[3];

for (int i = 0; i < 3; i++) {

printf("Enter details for student %d:\n", i + 1);

printf("Name: ");

fgets(students[i].name, sizeof(students[i].name), stdin);

students[i].name[strcspn(students[i].name, "\n")] = '\0';

printf("Roll Number: ");

scanf("%d", &students[i].rollNumber);

printf("Marks: ");

scanf("%f", &students[i].marks);

getchar();

}

printf("\nStudent Details:\n");

for (int i = 0; i < 3; i++) {

printf("\nStudent %d\n", i + 1);

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

printf("Roll Number: %d\n", students[i].rollNumber);

printf("Marks: %.2f\n", students[i].marks);

}

}

Q.13. File Handling in C

THEORY EXERCISE:

 Explain the importance of file handling in C. Discuss how to perform file

operations like opening, closing, reading, and writing files.

File handling in C allows programs to read from and write to files on disk. It is an essential feature because it enables the program to store data permanently, even after the program terminates. Without file handling, data would only exist during the program's runtime and would be lost once the program ends.

Key reasons for the importance of file handling in C:

1. **Persistence of Data**:
   * File handling allows programs to save information in files, making it possible to retain data even after the program stops running. For example, storing user preferences, application settings, or data logs.
2. **Data Sharing**:
   * Files provide a way for different programs or users to access and exchange data. File handling allows one program to write data to a file, and another program to read it.
3. **Data Backup**:
   * Programs can use file operations to back up important data to disk, ensuring that the data can be restored later, even in case of a crash or power failure.
4. **Large Data Handling**:
   * Files allow programs to handle large datasets that cannot fit entirely into memory. You can process large amounts of data incrementally, reading and writing portions of the data as needed.
5. **Interoperability**:
   * Files are commonly used to exchange data between different systems, platforms, and languages. File formats like CSV, JSON, and XML are widely used for data interchange.

**File Operations in C:**

File handling in C is performed using a set of standard library functions provided by the <stdio.h> header. The main operations involved in file handling are **opening**, **closing**, **reading**, and **writing** files. Below is a detailed explanation of each operation:

**1. Opening a File (fopen)**

To open a file, you use the fopen() function. It takes two arguments:

* The name of the file to be opened.
* The mode in which the file is opened (e.g., read, write, append).

Syntax:

FILE \*fopen(const char \*filename, const char \*mode);

Modes for opening files:

* "r": Open for reading. If the file doesn't exist, the program will fail.
* "w": Open for writing. If the file exists, it will be overwritten; if not, a new file is created.
* "a": Open for appending. Data will be written at the end of the file.
* "r+": Open for both reading and writing.
* "w+": Open for both reading and writing. Existing file is overwritten, and if the file doesn't exist, it will be created.
* "a+": Open for both reading and appending.

Example:

FILE \*file = fopen("data.txt", "r"); // Opens a file for reading

if (file == NULL) {

printf("Error opening file.\n");

}

**2. Closing a File (fclose)**

Once you are done working with a file, it is important to close it using the fclose() function. This releases the file pointer and ensures that all changes are saved.

Syntax:

int fclose(FILE \*file);

Example:

fclose(file); // Close the file after usage

**3. Reading from a File (fgetc, fgets, fread)**

There are several ways to read data from a file in C:

* **fgetc()**: Reads a single character from the file.

Syntax:

int fgetc(FILE \*file);

Example:

char ch;

ch = fgetc(file);

printf("Character read: %c\n", ch);

* **fgets()**: Reads a string (line) from the file. It is safer than scanf() because it limits the number of characters read.

Syntax:

char \*fgets(char \*str, int n, FILE \*file);

Example:

char str[100];

fgets(str, sizeof(str), file);

printf("Line read: %s\n", str);

* **fread()**: Reads a block of data from the file.

Syntax:

size\_t fread(void \*ptr, size\_t size, size\_t count, FILE \*file);

Example:

int buffer[10];

fread(buffer, sizeof(int), 10, file);

**4. Writing to a File (fputc, fputs, fwrite)**

There are several functions to write data to a file:

* **fputc()**: Writes a single character to the file.

Syntax:

int fputc(int char, FILE \*file);

Example:

fputc('A', file); // Writes 'A' to the file

* **fputs()**: Writes a string to the file.

Syntax:

int fputs(const char \*str, FILE \*file);

Example:

fputs("Hello, World!", file); // Writes the string to the file

* **fwrite()**: Writes a block of data to the file.

Syntax:

size\_t fwrite(const void \*ptr, size\_t size, size\_t count, FILE \*file);

Example:

int buffer[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

fwrite(buffer, sizeof(int), 10, file);

**5. File Error Handling (feof, ferror)**

You can check for errors while working with files using:

* **feof()**: Checks if the end of the file has been reached.

int feof(FILE \*file);

Example:

if (feof(file)) {

printf("End of file reached.\n");

}

* **ferror()**: Checks if an error occurred during file operations.

int ferror(FILE \*file);

Write a C program to create a file, write a string into it, close the file, thenopen the file again to read and display its contents.

int main() {

FILE \*file;

file = fopen("example.txt", "w");

if (file == NULL) {

printf("Error opening file for writing.\n");

}

fprintf(file, "Hello, this is a test string written into the file.\n");

fclose(file);

printf("File created and data written successfully.\n");

file = fopen("example.txt", "r");

if (file == NULL) {

printf("Error opening file for reading.\n");

}

char buffer[256];

printf("\nContents of the file:\n");

while (fgets(buffer, sizeof(buffer), file) != NULL) {

printf("%s", buffer); // Print each line read from the file

}

fclose(file);

}