MODULE 2

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?

- C was created by Dennis Ritchie at Bell Labs in 1972. It evolved from the B programming language, which itself was derived from BCPL (Basic Combined Programming Language). The main motivation behind the development of C was to enable the creation of the UNIX operating system in a more portable and efficient way than assembly language.
- As C gained popularity, especially in academia and industry, the need for standardization became apparent. In 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C. The result was ANSI C, also known as C89, which became the foundation for all modern C implementations.
- Importance of C:

1. Operating Systems: C remains the language of choice for developing operating systems. UNIX, Linux, and even major components of Windows are written in C.
2. Embedded Systems: Due to its efficiency and control over hardware, C is widely used in embedded programming, from microcontrollers to IoT devices.
3. Compilers and Interpreters: Many compilers for other languages are themselves written in C, including those for Python and Perl.
4. Portability and Performance: C programs can be compiled on a wide range of platforms with minimal changes, offering high performance with relatively low memory usage.

- Continued Relevance Today:
- Even in an era dominated by languages like Python, Java, and JavaScript, C remains crucial for several reasons:

1. Performance: C provides unmatched speed and minimal overhead, which is critical in system-level programming.
2. Foundation for Other Languages: Many modern languages are either implemented in C or inspired by it, including C++, Objective-C, Rust, and Go.
3. Educational Value: Learning C helps students understand core programming concepts like memory management, pointers, and data structures.

4. Wide Availability of Tools: The ecosystem of compilers, debuggers, and IDEs for C is mature and widely supported.

- 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.

- C programming is extensively used in embedded systems, operating systems, and game development. It provides the low-level control and efficiency needed for these applications. Embedded systems, like microcontrollers, rely on C for controlling real-time hardware, while operating systems like Linux use C for core functionality and device drivers. Game development also leverages C's performance for handling graphics and physics.

1. Embedded Systems (e.g., Automotive Control Systems):
    - Why C: C is close to hardware and offers high performance with low-level memory access, which is crucial in embedded systems where resources are limited.
    - Use Case: C is used to write firmware that controls sensors, actuators, and communication protocols like CAN (Controller Area Network).
    - Example: Engine Control Units (ECUs) in cars.
2. Operating Systems (e.g., Linux Kernel):
    - Why C: C provides a good balance between low-level control and portability, making it ideal for writing kernels and system-level code.
    - Use Case: Most parts of Unix/Linux systems (including file systems, memory management, and process control) are written in C.
    - Example: Linux OS and its kernel.
3. Game Development (e.g., Game Engines):
    - Why C: C (and often C++) offers high performance needed for real-time rendering and game logic.
    - Use Case: C is used in performance-critical parts of the game engine such as rendering pipelines, physics simulation, and audio processing.
    - Example: Unreal Engine (core engine components).

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?
    - To install a C compiler like GCC and an IDE like DevC++, VS Code, or Code::Blocks, you'll first install the compiler and then the IDE. For Windows, MinGW is a common choice for GCC. For IDEs, you'll download and install the appropriate software.

1. Install the GCC Compiler:

- **For Windows, use MinGW-w64:** Download the MinGW-w64 installer.
- **Extract and Run:** After downloading the installer, extract the contents and run the installation program.
- **Choose GNU Compilers:** During installation, select the appropriate GNU compilers package.
- **Set Environment Path:** Add the MinGW's bin directory (e.g., C:\\MinGW\\bin) to your system's PATH variable.

2. Install an IDE (DevC++, VS Code, or Code::Blocks):

- **DevC++:** Download and install DevC++ from the official website.
- **VS Code:** Download and install VS Code from the official website.
- **Code::Blocks:** Download and install Code::Blocks, either with MinGW or without it depending on your needs, from the official website.
- **For VS Code, install extensions:** Install the "C/C++" extension in VS Code.

3. Configure the IDE (if necessary):

- **VS Code:** In VS Code, you might need to configure the debugger settings and ensure your C/C++ extensions are

correctly set up to use your chosen compiler (e.g., GCC through MinGW).

- **Code::Blocks:** Code::Blocks usually auto-detects the MinGW compiler if you've installed it alongside the IDE. You might need to manually configure the compiler path if necessary.
- **DevC++:** DevC++ usually comes with GCC pre-installed, but you might need to configure the compiler path if you're using a different compiler.

4.(Optional) Add Compiler to PATH (Windows):

- Go to "Control Panel > System and Security > System > Advanced system settings > Environment Variables".
- Find "Path" under System Variables, edit it, and add the path to the MinGW's bin directory (e.g., C:\\MinGW\\bin).

5.Verify Installation:

- In the terminal or command prompt, type `gcc --version`. If GCC is correctly installed and the path is set, you should see the GCC version information.
- Open a new project in your IDE, write a simple "Hello, world!" program, and build it to confirm everything is working as expected.

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.

➢ A basic C program includes headers for standard functions, a `main` function as the entry point, comments for explanation, data types to store values, and variables to hold data.

1. Headers:

- **Purpose:** Headers (`.h` files) contain function declarations and other declarations (like data type definitions) that are used in the program.

- **Example:** `stdio.h` provides input/output functions like `printf` and `scanf`.

2.Main Function:

- **Purpose:** The main function is the entry point where the program execution begins.
- Example:

```
int main() { // Main function (return type int)
    // Code within main function
    return 0; // Indicates successful execution
}
```

3.Comments:

- **Purpose:** Comments explain the code and are ignored by the compiler.
- **Types:**
  Multi-line comments: Start with /* and end with */.
  Single-line comments: Start with //.
- Example:

```
/*
    This is a multi-line comment
    explaining the program.
    */
    // This is a single-line comment
```

4. Data Types:

- **Purpose:** Data types define the type of data a variable can store (e.g., integers, floating-point numbers, characters).
- **Common types:**
  `int`: Stores integer values (e.g., 10, -5).
  `float`: Stores floating-point values (e.g., 3.14).
  `char`: Stores single characters (e.g., 'A', '!').
- Example:

```
int age = 25; // Integer variable
    float price = 19.99; // Floating-point variable
    char grade = 'A'; // Character variable
```

5.Variables:

- **Purpose:** Variables are memory locations that store values.
- **Declaration:** Variables are declared using a data type and a name.
- Example:

```
int number; // Declares an integer variable named 'number'
```

number = 10; // Assigns the value 10 to the variable 'number'

- Complete Example:

```
#include <stdio.h> // Include standard input/output library

int main() { // Main function
    int age = 25; // Declare and initialize an integer variable 'age'
    float price = 19.99; // Declare and initialize a float variable 'price'
    char grade = 'A'; // Declare and initialize a char variable 'grade'

    printf("Age: %d\n", age); // Print the value of 'age'
    printf("Price: %f\n", price); // Print the value of 'price'
    printf("Grade: %c\n", grade); // Print the value of 'grade'

    return 0; // Indicates successful execution
}
```

Operators in C

- THEORY EXERCISE:

Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.

1. Arithmetic Operators:
   Used to perform basic mathematical operations:

Operator	Meaning	Example
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	A*b
/	Division	a/b
%	Modulus(reminder)	a%b

2.Relational Operators:
Used to compare two values:

Operator	Meaning	Example

==	Equal to	a == b
!=	Not equal to	a != b
>	Greater Than	a > b
<	Less Than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a<= b

3.Logical Operators:

Used to perform logical operations:

Operator	Meaning	Example
&&	Logical AND	a>0 && b>0
`		`
!	Logical NOT	! (a>b)

4. Assignment Operators:

Operator	Meaning	Example
=	Assign	a = 5
+=	Add and assign	a += 5(a=a+5)
-+	Subtract and assign	a -+ 5
*=	Multiply and assign	a *= 5
/=	Divide and assign	a /= 5
%=	Modulus and assign	a%=5

5. Increament and Decrement Operators:
Used to increase or decrease values by 1:

Operator	Meaning	Example
++	Increament by 1	a++ or ++a
--	Decrement by 1	a—or –a

- ++a Pre-increment (increments then uses).
- a++ Post-increment (uses then increments).

6. Bitwise Operators:
   Used to perform bit-level operations:

Operator	Meaning	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT (1's complement)	~a
>>	Right shift	a >> 1
<<	Left shift	a << 1

7. Conditional (Ternary) Operator:
   Used to evaluate a condition in a single line:

Operator	Meaning	Example
?:	Ternary (if-else shortcut)	condition ? expr1 : expr2

Control Flow Statements in C

- THEORY EXERCISE:

❖ **Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.**

1. If Statement

   Executes a block of code only if a specified condition is true

   - Syntax:

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

   - Example:

```
int num = 10;
if (num > 0) {
    printf("Number is positive.\n");
}
```

2. If-else Statement:

   Executes one block of code if the condition is true and another if it's false.

   - Syntax:

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

```
}
```

- Example:

```
int num =-5;
if (num >= 0) {
   printf("Number is positive.\n");
} else {  printf("Number is negative.\n"}
```

3. Nested if-else Statement:
    - Syntax:

```
if (condition1) {
   // code
} else {
   if (condition2) {
      // code
   } else {
      // code
   }
}
```

    - Example:

```
int num = 0;
if (num > 0) {
   printf("Positive number\n");
} else if (num < 0) {
   printf("Negative number\n");
} else {
   printf("Zero\n");
```

4. Switch Statement:
    - Syntax:

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

    - Example:

```
int day = 3;
switch (day) {
   case 1:
      printf("Monday\n");
      break;
```

```
    case 2:
        printf("Tuesday\n");
        break;
    case 3:
        printf("Wednesday\n");
        break;
    default:
        printf("Invalid day\n");
}
```

Looping in C.

- THEORY EXERCISE:

❖ Explain the use of break, continue, and go to statements in C. Provide examples of each.

➢ In C programming, break, continue, and go to are jump statements that alter the normal flow of execution. break exits a loop or switch statement. continue skips the current iteration of a loop and moves to the next one. Go to transfers control to a labelled statement within the same function.

1. break statement:

- **Purpose:** Terminates the execution of a loop (for, while, do-while) or a switch statement prematurely. Control then jumps to the statement immediately following the loop or switch statement.
- **Example:**

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // exits the loop when i is 5
}
        printf("%d ", i);
}
    return 0;
}
```

Output: 1 2 3 4

2. continue statement:

- **Purpose:** Skips the remaining statements in the current iteration of a loop and proceeds to the next iteration. The loop condition is checked, and if true, the next iteration begins.
- **Example**:

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

```
int main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3) {
         continue;  // skips printing when i is 3
}
      printf("%d ", i);
}
   return 0;
}
```

Output: 1 2 4 5

3. go to statement:
   - **Purpose:** Transfers control to a labeled statement within the same function. Labels are defined by an identifier followed by a colon (`:`). `goto` is often used to implement complex control flow, but its overuse can make code harder to read and maintain.
   - **Example:**

```
#include <stdio.h>

int main() {
    int i = 1;
    start:
    printf("%d ", i);
    i++;
    if (i <= 5)
        goto start;  // jumps back to the label
return 0;
}
```

Output: 1 2 3 4 5

Functions in C.

- THEORY EXERCISE:

❖ What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

➢ In C programming, functions are blocks of code that perform specific task. They help in breaking down complex problems into smaller, manageable parts, improve code reusability and make debugging easier.

1. Function Declaration (or Prototype)
   It tells the compiler about the function's name, return type, and parameters before its actual use.
   ❖ Syntax:

return_type function_name(parameter_list);

- ❖ Example:
  Int add(int a, int b);

2. Function Definition:
   This is where the actual code of the function is written. It must match the declaration in type and parameters.
   - ❖ Syntax:
     return_type
     function_name(parameter_list) {
     // function body
     }
   - ❖ Example:
     Int add(int a, int b) {
     return a+b;
     }
3. Function Call:
   To use the function, call it by its name and pass required arguments.
   - ❖ Syntax:
     Function_name(arguments);

Arrays in C.

- THEORY EXERCISE:

❖ Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

- ➢ An array in C is a collection of elements of the same data type stored in contiguous memory locations. It allows storing multiple values using a single variable name, accessed using an index.
- ❖ Syntax:
  data_type array_name[size];
- ❖ Example:
  Int number [5]; // Declares an array of 5 integers.

- Difference between One-Dimensional and Multi-Dimensional Arrays :

Feature	One-Dimensional	Multi-Dimensional
Structure	Linear, single row	Table-like (2D), Cube-like (3D), etc.
Declaration Syntax	data_type array_name [size]	data_type array_name [size] [size2]; (2D)
Access Method	Array [index]	Array [row] [column] (2D example)
Use Case	List of items (e.g., marks of students)	Matrix operations, tables (e.g., seating chart)

- One-Dimensional Array Example:

```
#include <stdio.h>
int main() {
   int marks[5] = {90, 85, 78, 92, 88};
   for(int i = 0; i < 5; i++) {
      printf("Marks[%d] = %d\n", i, marks[i]);
}
   return 0;
}
```

- Multi-Dimensional Array Example:

```
#include<stdio.h>
int main(){
   int matrix[2][3] = {
      {1, 2, 3},
      {4, 5, 6}
};
   for(int i = 0; i < 2; i++) {
      for(int j = 0; j < 3; j++) {
```

```
            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
        } return 0; }
```

Pointers in C

- THEORY EXERCISE:

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

➢ A pointer in C is a variable that stores the memory address of another variable. Pointers allow for powerful and flexible programming techniques such as dynamic memory allocation, efficient array handling, and passing large structures or arrays to functions without copying.

➢ Declaration and Initialization –

❖ Syntax:

```
data_type *pointer_name;
```

❖ Example:

```
Int x = 10;
Int *p = &x;  // p stores the address of variable x.
```

❖ *p is the dereferenced value (i.e., the value stored at the address)

❖ &x gives the address of variable x.

❖ Example:

```
#include <stdio.h>

int main() {
   int a = 5;
   int *ptr = &a;

   printf("Value of a: %d\n", a);
   printf("Address of a: %p\n", &a);
   printf("Value stored in ptr (address of a): %p\n", ptr);
   printf("Value pointed to by ptr: %d\n", *ptr);

   return 0;
}
```

❖ Pointers are important in C:

1. Efficient Memory Management:
   Enable dynamic memory allocation using malloc, calloc, free.
2. Function Argument Passing:
   Allow functions to modify actual arguments (pass-by-reference).
3. Efficient Array Handling:
   Arrays and pointers are closely related; pointers can iterate through arrays efficiently.
4. Data Structures:
   Essential for creating linked lists, trees, graphs, etc.

5. Hardware and Low-Level Access:
   Pointers enable direct memory and hardware register access (useful in systems programming).

String 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, like strlen(), strcpy(), strcat(), strcmp(), and strchr(), are essential for manipulating text in programs. strlen() gets the length of a string, strcpy() copies one string to another, strcat() concatenates (joins) strings, strcmp() compares strings, and strchr() searches for the first occurrence of a character. These functions simplify tasks like determining string length, copying data, joining text, and searching for specific characters within strings.

1. strlen():
   - Purpose: Calculates the length of a string, returning the number of characters before the null terminator (\0).
   - Example: int length = strlen("Hello, world!"); would return 13.
   - Usefulness: Determining the size of a string before allocating memory, checking if a string is empty, or working with array indices.
2. strcpy():
   - Purpose:
     Copies a string (including the null terminator) from one location (source) to another (destination).
   - Example:
     strcpy(destination, "Hello"); would copy the string "Hello" into the destination array.
   - Usefulness:
     Storing user input, initializing string variables, or transferring data between different memory locations. Important: Ensure the destination has enough space to hold the copied string to prevent buffer overflows.
3. strcat():
   - Purpose: Appends one string to the end of another.
   - Example: strcat(string1, string2); appends string2 to the end of string1.
   - Usefulness: Building up strings from smaller parts, creating user-friendly output, or formatting text. Important: Ensure the destination string has enough space to accommodate the appended string.
4. strcmp():
   - Purpose: Compares two strings lexicographically (alphabetically).

- Example: int result = strcmp("apple", "banana"); would return a negative value (since "apple" comes before "banana").
- Usefulness: Sorting strings, searching for specific strings, or comparing user input against expected values.

5. strchr():
    - Purpose: Searches for the first occurrence of a specific character within a string.
    - Example: char* found = strchr("Hello, world!", 'o'); would return a pointer to the first 'o' in the string.
    - sefulness: Finding the position of a delimiter, splitting a string, or checking if a character is present in a string.

Structures in C.

- THEORY EXERCISE:

❖ Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

➢ Concept of Structures in C:
In C, a structure is a user-defined data type that allows grouping variables of different types under one name. Structures are used to represent a record, such as a student, employee, or product, where various related data types need to be grouped together.

➢ Declaring a Structure:
You define a structure using the struct keyword followed by the structure name and a block containing member definitions.

- Syntax:

Struct Student {
int id;
char name [50];
float marks;
};
This defines the structure named Student with three members id, name, and marks.

➢ Initializing Structure Members:
You can initialize a structure at the time of declaration.

- Syntax:
  Struct Student s1 = {101, “Alice”, 85.5};
  Or assign values individually:
  s1.id = 101;

```
strcpy(s1.name, “Alice”);  // Use strcpy for strings
s1.marks = 85.5
```

- Accessing Structure Members:
  Use the dot operator (.) to access members.
  - Syntax:
    ```
    printf(“ID: %d\n”, s1.id);
    Printf(“Name: %s\n”, s1.name);
    Printf(“Marks: %.2f\n”, s1.marks);
    ```

If using a pointer to a structure, use the arrow operator (->):

- Syntax:
  ```
  Struct Student *ptr = &s1;
  Printf(“ID: %d\n”, ptr->id);
  ```

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.
    - Importance of File Handling in C:
      File handling in C is crucial for managing data storage and retrieval beyond the runtime of a program. Unlike variables and arrays, which lose their data once a program terminates, files allow for persistent storage of information. File handling enable:
      1. Data Persistence: Data can be stored permanently.
      2. Large Data Management: Files can handle large amounts of data efficiently.
      3. Data Sharing: Data in files can be shared across different programs or systems.
      4. Backup and Recovery: Files serve as a backup source in case of data loss in memory.
    - File Operations in C:
      1. Opening a File:

- The fopen() function is used to open a file, specifying the filename and the desired mode (e.g., read, write, append).
- Example: FILE *fp; fp = fopen("filename.txt", "r"); (Opens "filename.txt" in read mode).

2. Closing a File:
    - The fclose() function closes a file after it's no longer needed, releasing resources and ensuring data is properly saved.
    - Example: fclose(fp); (Closes the file pointer fp).
3. Reading from a File:
    - Various functions can be used to read data from a file, including fgetc() (reads a single character), fgets() (reads a line), and fscanf() (reads formatted data).
    - Example: char ch = fgetc(fp); (Reads a character from file fp).
4. Writing to a File:
    - Functions like fputc() (writes a single character), fputs() (writes a string), and fprintf() (writes formatted data) can be used to write data to a file.
    - Example: fputc('A', fp); (Writes the character 'A' to file fp)