Module 2 – Introduction to Programming

- ## Overview of C Programming

Q.1 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 programming was developed by Dennis Ritchie at Bell Labs in 1972 as an improvement over the B language. It was first used to rewrite the UNIX operating system, which made UNIX portable and widely adopted. This marked a turning point in software development, proving that operating systems could be written in a high-level language.

In 1978, Brian Kernighan and Ritchie published *"The C Programming Language"*, which helped popularize C. Later, the ANSI C standard (1989) and further updates like C99 and C11 ensured that C remained modern and consistent across platforms.

C is important because it combines efficiency, portability, and control over hardware. It is the foundation for many modern languages such as C++, Java, and Python, and remains widely used in operating systems, embedded systems, device drivers, and compilers.

Even today, C is valued for its speed, reliability, and educational importance. It teaches programmers how computers work at a deeper level and continues to power critical technologies worldwide.

The evolution of C shows how a language from the 1970s still shapes modern computing. Its balance of low-level control and high-level structure ensures that C remains one of the most important programming languages in history.

Q.2 Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.

Ans: - There are three concrete, high-impact places where C is used extensively—what they are, why C fits, and real products that rely on it:

1. Operating systems (kernels)

- What: Core kernel code that manages memory, processes, filesystems, and device drivers.

- Why C: Precise control over memory and hardware with predictable performance, yet portable across architectures.
- Real examples: The Linux kernel is primarily written in C (GNU C11), and powers everything from servers to Android devices.

2. Embedded & IoT firmware (microcontrollers)

- What: Software for tiny devices—sensors, wearables, industrial controllers, automotive ECUs.
- Why C: Tiny RAM/flash budgets, direct register access, determinism, and broad compiler/toolchain support.
- Real examples: FreeRTOS (a popular RTOS for microcontrollers) is supplied as standard C source and is "mostly written in C," widely used on ARM Cortex-M (e.g., STM32). ST's STM32Cube tools generate C initialization code for these chips.

3. Databases & storage engines

- What: High-performance, embeddable data engines used inside apps, phones, and browsers.
- Why C: Small footprint, speed, and easy embedding as a library.
- Real examples: SQLite—a self-contained SQL database—is a C-language library and is the world's most widely deployed database, built into all mobile phones and many desktop apps.

• Setting Up Environment

Q.1 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.

Ans: - **1. Installing GCC (C Compiler)**

GCC (GNU Compiler Collection) is the most widely used compiler for C.
Windows (via MinGW or TDM-GCC)

1. Download MinGW or TDM-GCC from their official sites.
2. Run the installer and select the gcc component.
3. During installation, note the folder path (e.g., C:\MinGW\bin).
4. Add this path to your System Environment Variables → PATH.
   - Search "Environment Variables" → Edit PATH → Add C:\MinGW\bin.
5. Verify installation: Open Command Prompt and type:
6. gcc --version

If it shows version info, GCC is installed successfully.
Linux (Ubuntu/Debian)
sudo apt update
sudo apt install build-essential
gcc --version
MacOS

1. Install Xcode Command Line Tools:
2. xcode-select --install
3. Verify with gcc --version.

2. Installing and Setting up an IDE
(a) Dev-C++

1. Download Dev-C++ (Orwell or Embarcadero version).
2. Run installer and launch IDE.
3. By default, Dev-C++ comes with a compiler (MinGW).
4. Create a new C project → Write code → Press F9 to Compile & Run.

(b) Code::Blocks

1. Download Code::Blocks with MinGW setup (choose the installer that includes MinGW).
2. Install it → the compiler will be auto-configured.
3. Open Code::Blocks → New Project → Console Application → Select C.
4. Write code → Press F9 to Build & Run.

(c) Visual Studio Code (VS Code)

1. Download and install VS Code.

2. Install the C/C++ extension (Microsoft).
3. Ensure GCC (MinGW) is installed and added to PATH.
4. In VS Code:
    - Create a new folder for your project.
    - Open it in VS Code → Create hello.c.
    - Write your code.
5. Open Terminal in VS Code → Compile with:
6. gcc hello.c -o hello
7. ./hello
8. (Optional) Configure tasks.json to run builds with one click.

3. Installing Turbo C++ (for Windows)

Since Turbo C++ is a DOS-based compiler, it doesn't run natively on modern Windows. We usually run it through DOSBox or use a pre-packaged installer.
Steps:

1. Download Turbo C++ setup (usually packaged with DOSBox, like "Turbo C++ for Windows 7/8/10" installer).
2. Run the installer → it automatically configures Turbo C++ inside DOSBox.
3. After installation, you'll see a desktop shortcut (e.g., *TurboC3*).
4. Launch it → The classic Turbo C++ IDE (blue screen) will appear.
5. To write and run a program:
    - Open File → New → Type your C program.
    - Save it with .C extension (e.g., hello.c).
    - Go to Compile → Compile (or press Alt + F9).
    - Then Run → Run (or press Ctrl + F9).

IDE	**Compiler Used**	**Best For**
Turbo C++	Borland Turbo C	Old syllabus, DOS-based programs, academics
Dev-C++	MinGW (GCC)	Beginners, simple lightweight environment
Code::Blocks	GCC/MinGW	Easy setup with projects & debugging
VS Code	GCC/Clang/MSVC	Modern, extensions, professional projects

• Basic Structure of a C Program

Q.1 Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.

Ans: - A C program always follows a structured format. Here are the main parts:

1. Header Files

- Header files provide predefined functions (like printf, scanf, etc.).
- Common headers:

```
    #include <stdio.h> → for input/output

    #include <stdlib.h> → for memory and utility functions

    #include <math.h> → for math functions
```

Example:

```
#include <stdio.h>   // standard input-output header
```

2. Comments

- Used to explain code, ignored by the compiler.
- Single-line: // comment here
- Multi-line:

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

3. main() Function

- Every C program starts execution from main().
- Syntax:

```
int main() {
   // code
   return 0;   // indicates program ended successfully
}
```

4. Data Types

- Define the kind of data a variable can hold.

- Common data types:
    - int → integers (e.g., 10, -5)
    - float → decimal numbers (e.g., 3.14)
    - char → single characters (e.g., 'A')
    - double → double-precision floating numbers

5. Variables

- Variables are named memory locations to store data.
- Syntax:
- data_type variable_name = value;

Example:

int age = 20;

float pi = 3.14;

char grade = 'A';

- **Headers**: provide built-in functions
- **Comments**: explain the code
- **main()**: entry point of program
- **Data types**: define type of data (int, float, char, etc.)
- **Variables**: store values for use in program

Operators in C

Q.1 Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.

Ans: -

Operators in C

Operators are special symbols used to perform operations on variables and values. In C, operators are grouped into categories:

1. Arithmetic Operators

- Used for mathematical calculations.
- Operators: +, -, *, /, %
- Examples:

```
int a = 10, b = 3;

printf("%d\n", a + b);  // 13 (Addition)

printf("%d\n", a - b);  // 7  (Subtraction)

printf("%d\n", a * b);  // 30 (Multiplication)

printf("%d\n", a / b);  // 3  (Division - quotient)

printf("%d\n", a % b);  // 1  (Modulus - remainder)
```

2. Relational Operators

- Compare two values, result is either true (1) or false (0).
- Operators: ==, !=, <, >, <=, >=
- Example:

```
int x = 5, y = 10;

printf("%d\n", x < y);   // 1 (true)

printf("%d\n", x == y);  // 0 (false)
```

3. Logical Operators

- Combine conditions, result is true (1) or false (0).
- Operators:

  - && (Logical AND) → true if both conditions are true
  - || (Logical OR) → true if at least one condition is true
  - ! (Logical NOT) → reverses the result
- Example:

```
int a = 5, b = 10;
printf("%d\n", (a < b) && (b > 0)); // 1 (true)
printf("%d\n", (a > b) || (b > 0)); // 1 (true)
printf("%d\n", !(a < b));         // 0 (false)
```

4. Assignment Operators

- Assign values to variables.
- Operators: =, +=, -=, *=, /=, %=
- Example:

```
int n = 10;
n += 5;   // n = n + 5 → 15
n -= 3;   // n = n - 3 → 12
```

5. Increment and Decrement Operators

- Increase or decrease a value by 1.
- Operators: ++, --
- Types:
  - Pre-increment ++a → increment, then use value
  - Post-increment a++ → use value, then increment
- Example:

```
int a = 5;
printf("%d\n", ++a); // 6 (first increment, then print)
printf("%d\n", a++); // 6 (print, then increment → a becomes 7)
```

6. Bitwise Operators

- Work at the bit level (binary operations).
- Operators:
  - & (AND)
  - | (OR)
  - ^ (XOR)
  - ~ (NOT)
  - << (Left shift)
  - >> (Right shift)
- Example:

```
int x = 5, y = 3;  // 5 = 0101, 3 = 0011
printf("%d\n", x & y);  // 1 (0001)
printf("%d\n", x | y);  // 7 (0111)
printf("%d\n", x ^ y);  // 6 (0110)
printf("%d\n", x << 1); // 10 (1010)
printf("%d\n", x >> 1); // 2  (0010)
```

7. Conditional (Ternary) Operator

- Shorthand for if-else.
- Syntax:
- condition ? expression1 : expression2;
- Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
printf("Max = %d\n", max); // Max = 20
```

- ## Control Flow Statements in C

Q.1 Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

Ans: - Decision-making statements allow the program to take different actions depending on conditions.

1. if Statement

- Used to test a condition. If true, the block runs.
- Syntax:

```
if (condition) {
   // code executes if condition is true
}
```

Example:

```
int age = 18;
if (age >= 18) {
   printf("You are eligible to vote.\n");
}
```

Output:

```
You are eligible to vote.
```

2. if-else Statement

- Provides two paths: one for true, one for false.
- Syntax:

```
if (condition) {
   // executes if true
} else {
   // executes if false
}
```

- Example:

```
int num = 5;
```

```
if (num % 2 == 0) {
   printf("Even number\n");
} else {
   printf("Odd number\n");
}
```

Output:

```
Odd number
```

3. Nested if-else

- An if-else inside another if-else.
- Useful for multiple conditions.
- Example:

```
int marks = 75;
if (marks >= 90) {
   printf("Grade A\n");
} else if (marks >= 75) {
   printf("Grade B\n");
} else if (marks >= 50) {
   printf("Grade C\n");
} else {
   printf("Fail\n");
}
```

Output:

```
Grade B
```

4. switch Statement

- Used when you have multiple choices for one variable.
- More readable than long if-else chains.
- Syntax:

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // code if no case matches
}
```

- Example:

```
int day = 3;
switch (day) {
  case 1: printf("Monday\n"); break;
  case 2: printf("Tuesday\n"); break;
  case 3: printf("Wednesday\n"); break;
  case 4: ("Thursday\n"); break;
  case 5: printf("Friday\n"); break;
  default: printf("Weekend\n");
}
```

Output:

```
Wednesday
```

• Looping in C

Q.1 Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.

Ans:

Loops allow repeating a block of code multiple times until a condition is met.
C provides while, for, and do-while loops.

1. while Loop

- Syntax:

```
while (condition) {
  // code to execute
}
```

- How it works:
  - Condition is checked before execution.
  - If condition is false at the start → loop does not run even once.
- Use-case: When the number of iterations is unknown in advance, but depends on a condition.
- Example:

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

2. for Loop

- Syntax:
- for (initialization; condition; update) {
-   // code to execute
- }
- How it works:
  - Best when the number of iterations is known.

    - Initialization, condition-check, and update all in one line.

- Use-case: Counting, iterating arrays, fixed repetitions.
- Example:

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

3. do-while Loop

- Syntax:

```
do {
  // code to execute
} while (condition);
```

- How it works:
    - Code runs at least once, since condition is checked after execution.
- Use-case: When you need the loop body to execute at least once, like menu-driven programs or input validation.
- Example:

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

- ## Loop Control Statements

Q.1 Explain the use of break, continue, and goto statements in C. Provide examples of each.

Ans: - Sometimes we need to alter the normal flow of loops. C provides three main control statements: break, continue, and goto.

1. break Statement

- Purpose: Exits immediately from the loop (or switch statement), regardless of the condition.
- Use-case: When you find the result early and don't need further iterations.
- **Example**:

```
#include <stdio.h>

int main() {

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

    if (i == 5) {

      break;  // loop ends when i == 5

    }

    printf("%d ", i);

  }

  return 0;

}
```

Output:

```
1 2 3 4
```

2. continue Statement

- Purpose: Skips the current iteration and jumps to the next iteration of the loop.
- Use-case: When you want to ignore some values but continue looping.
- **Example**:

```
#include <stdio.h>

int main() {

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

Output:

```
1 2 4 5
```

3. goto Statement

- Purpose: Transfers control to a labeled statement in the program.
- Use-case: Rarely used (not recommended) → can make programs hard to read ("spaghetti code"), but sometimes useful for breaking out of deeply nested loops.
- **Syntax**:

```
goto label;
...
label:
    // code here
```

- **Example**:

```
#include <stdio.h>
int main() {
    int i = 1;
    start:   // label
    if (i <= 5) {
        printf("%d ", i);
        i++;
        goto start;  // jump back to label
    }
    return 0;
}
```

Output:

```
1 2 3 4 5
```

- ## Functions in C

Q.1 What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

Ans: A function is a block of code that performs a specific task.

- Functions help in:
    - Code reusability (write once, use many times)
    - Modularity (program divided into smaller parts)
    - Readability & debugging

In C, programs always start from the main() function, and other user-defined functions can be created.

Parts of a Function

1. Function Declaration (Prototype)

- Tells the compiler about the function name, return type, and parameters.
- Written before main().
- Syntax:
- return_type function_name(parameter_list);
- Example:
- int add(int a, int b);

2. Function Definition

- Actual body of the function → contains the code to be executed.
- Syntax:

```
return_type function_name(parameter_list) {
   // function body
   return value;
}
```

Example:

```
int add(int a, int b) {
   return a + b;
}
```

3. Function Call

- Tells the program to execute the function.
- Syntax:
- function_name(arguments);
- Example (inside main()):
- int sum = add(5, 3);   // calling add function

Complete Example Program

```
#include <stdio.h>
// Function declaration (prototype)
int add(int a, int b);
// main function
int main() {
   int x = 10, y = 20, result;
   // Function call
   result = add(x, y);
   printf("Sum = %d\n", result);
   return 0;
}
// Function definition
int add(int a, int b) {
   return a + b;   // returns sum to the caller
}
```

Output

```
Sum = 30
```

Arrays in C

Q.1 Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

Ans: An array is a collection of elements of the same data type, stored in contiguous memory locations.

- Instead of declaring many variables (int a1, a2, a3...), we can use arrays.
- Each element is accessed using an index (starting from 0).

Syntax:

```
data_type array_name[size];
```

1. One-Dimensional Array

Stores elements in a **single row** (like a list).

- **Declaration:**

```
int marks[5];    // array of 5 integers
```

- **Initialization:**

```
int marks[5] = {85, 90, 75, 88, 95};
```

- **Accessing Elements:**

```
printf("%d", marks[2]);  // prints 75 (index 2)
```

- **Example Program:**

```
#include <stdio.h>

int main() {

  int marks[5] = {85, 90, 75, 88, 95};

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

    printf("marks[%d] = %d\n", i, marks[i]);

  }

  return 0;

}
```

Output:

```
marks[0] = 85

marks[1] = 90
```

```
marks[2] = 75

marks[3] = 88

marks[4] = 95
```

2. Multi-Dimensional Arrays

- Arrays with two or more dimensions.
- Most common: 2D array (like a table with rows and columns).

2D Array Example

- **Declaration:**

```c
int matrix[2][3];   // 2 rows, 3 columns
```

- **Initialization:**

```c
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
```

- **Accessing Elements:**

```c
printf("%d", matrix[1][2]);  // prints 6
```

- **Example Program:**

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

Output:

```
1 2 3

4 5 6
```

Difference: One-Dimensional vs Multi-Dimensional Arrays

Feature	**1D Array**	**2D (Multi-Dimensional) Array**
Structure	Single row (linear list)	Rows and columns (table format)
Syntax	int arr[5];	int arr[3][4];
Access	arr[index]	arr[row][col]
Example Use	Storing marks of students	Storing marks in multiple subjects

- ## Strings in C

Q.1 Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.

Ans: - In C, strings are arrays of characters ending with a null character '\0'.
C provides many library functions (from <string.h>) to manipulate strings easily.

1. strlen() – String Length

- Returns the length of a string (number of characters, excluding '\0').

Syntax:

```
int strlen(const char *str);
```

Example:

```
#include <stdio.h>

#include <string.h>

int main() {

   char name[] = "Hello";

   printf("Length = %lu\n", strlen(name));

   return 0;

}
```

Output:

```
Length = 5
```

2. strcpy() – Copy String

- Copies one string into another.
- Be careful: destination must have enough space.

Syntax:

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

Example:

```
#include <stdio.h>

#include <string.h>

int main() {
  char src[] = "C Programming";
  char dest[50];
  strcpy(dest, src);
  printf("Copied String: %s\n", dest);
  return 0;
}
```

Output:

```
Copied String: C Programming
```

3. strcat() – Concatenate Strings

- Appends (joins) one string at the end of another.

Syntax:

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

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
  char s1[50] = "Hello ";
  char s2[] = "World!";
  strcat(s1, s2);
  printf("Concatenated: %s\n", s1);
  return 0;
}
```

Output:

```
Concatenated: Hello World!
```

4. strcmp() – Compare Strings

- Compares two strings lexicographically.
- Returns:

  - 0 → if strings are equal
  - <0 → if first string < second string
  - >0 → if first string > second string

Syntax:

```
int strcmp(const char *s1, const char *s2);
```

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
   char a[] = "apple";
   char b[] = "banana";
   int result = strcmp(a, b);
   if (result == 0) printf("Strings are equal\n");
   else if (result < 0) printf("a is smaller\n");
   else printf("a is greater\n");
   return 0;
}
```

Output:

```
a is smaller
```

5. strchr() – Find Character in String

- Finds the first occurrence of a character in a string.
- Returns a pointer to the character (or NULL if not found).

Syntax:

```
char* strchr(const char *str, int c);
```

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
   char str[] = "programming";
```

```
    char *ptr = strchr(str, 'g');
    if (ptr) printf("Found at position: %ld\n", ptr - str);
    else printf("Not found\n");
    return 0;
}
```

Output:

```
Found at position: 3
```

Function	**Purpose**	**Example Use**
strlen()	Get string length	Password length check
strcpy()	Copy string	Store user input in buffer
strcat()	Join strings	Make full file path "C:/Users/" + "Docs"
strcmp()	Compare strings	Login authentication
strchr()	Find character	Locate @ in email