Module 2 – Introduction to Programming
Overview of C Programming

1) Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

History & Evolution

- **Created in 1972** by **Dennis Ritchie** at Bell Labs.
- Developed to rewrite the **UNIX operating system** (originally in assembly).
- Evolved from earlier languages **BCPL** and **B**.
- **UNIX rewritten in C** → proved C's power.
- **ANSI C (1989)** → standard version of C.
- Later versions: **C99, C11** added modern features.

Importance of C

- **Foundation language** for C++, Java, and many others.
- Used in **system programming** (OS, compilers, embedded systems).
- **Portable** → runs on different machines easily.
- **Fast and efficient** → closer to hardware.

Why Still Used Today

1. High performance and speed.
2. Used in **operating systems, device drivers, robotics, databases**.
3. Good for learning **core programming concepts** (loops, arrays, pointers).
4. Reliable for low-level programming.

Conclusion

C is the **backbone of modern programming**, still popular because of its **speed, portability, and influence** on other languages.

2) 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

1. Install a C Compiler (GCC)

- **Download GCC**: Go to the official MinGW (Windows) or GCC website.
- **Install**: Run the installer and select "C Compiler" during installation.
- **Add to PATH**: Add the GCC bin folder path to the system environment variables so you can use gcc command in terminal.
- **Verify**: Open Command Prompt/Terminal → type gcc --version → if version shows, installation is successful.

2. Install an IDE

You can choose **DevC++**, **VS Code**, or **CodeBlocks**.

(a) DevC++

- Download DevC++ setup file.
- Install and open it.
- Write a new C program → press **Compile & Run** → it will use its built-in compiler.

(b) VS Code

- Download and install **Visual Studio Code**.
- Install the extension **C/C++ (by Microsoft)**.
- Set up tasks.json to use GCC for compiling.
- Open a folder, create a .c file, and run it using terminal commands or the extension.

(c) Code::Blocks

- Download **Code::Blocks with MinGW** (includes GCC).
- Install and open it.
- Create a new C project → write code → press **Build & Run**.

Conclusion

- **Compiler (GCC)** is needed to convert C code into machine code.
- **IDE (DevC++, VS Code, CodeBlocks)** makes coding easier with features like syntax highlighting, error checking, and one-click run.

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

Basic Structure of a C Program

A C program has a simple structure made of different parts:

1. Header Files

- Written at the top using #include.
- They allow us to use standard functions (like printf).
- Example:
  - #include <stdio.h>

2. Main Function

- Every C program starts from main().
- Code inside main() runs first.
- Example:

```
int main() {

  // code here
  return 0;
  }
```

3. Comments

- Used to explain the code, ignored by compiler.
- Types:
  - Single line: // This is a comment
  - Multi-line: /* This is a comment */

4. Data Types

- Define the type of data a variable can store.
- Examples:
  - int → integers (10, -5)
  - float → decimal numbers (3.14)
  - char → characters ('A', 'b')
  - double → large decimal numbers

5. Variables

- Named storage in memory to hold data.

- Example:

```
int age = 20;
float price = 99.5;
char grade = 'A';
```

Example Program

```
#include <stdio.h>   // Header file

// This program shows basic structure
int main() {
    int age = 20;          // integer variable
    float marks = 85.5;    // float variable

    printf("Age: %d\n", age);
    printf("Marks: %.2f\n", marks);

    return 0; // end of program
}
```

Output:

```
Age: 20
Marks: 85.50
```

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

Operators in C

Operators are symbols used to perform operations on variables and values.

1. Arithmetic Operators

Used for mathematical calculations.

- + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus → remainder).
- Example:

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

printf("%d", a % b);  // Output: 1
```

2. Relational Operators

Used to compare values, result is **true (1)** or **false (0)**.

- == (equal), != (not equal), > (greater), < (less), >=, <=.
- Example:

```
int x = 5, y = 10;
printf("%d", x < y);  // Output: 1 (true)
```

3. Logical Operators

Used with conditions, result is true (1) or false (0).

- && (AND), || (OR), ! (NOT).
- Example:

```
int a = 1, b = 0;
printf("%d", a && b);  // Output: 0
```

4. Assignment Operators

Used to assign values.

- = (simple assign), +=, -=, *=, /=, %=.
- Example:

```
int a = 5;
a += 3;   // same as a = a + 3 → a = 8
```

5. Increment/Decrement Operators

Used to increase or decrease value by 1.

- ++ (increment), -- (decrement).
- Example:

```
int a = 5;
a++;  // a = 6
--a;  // a = 5 again
```

6. Bitwise Operators

Work on bits (0 and 1).

- & (AND), | (OR), ^ (XOR), ~ (NOT), << (Left shift), >> (Right shift).
- Example:

```
int a = 5, b = 3;
printf("%d", a & b);  // Output: 1 (0101 & 0011 = 0001)
```

7. Conditional (Ternary) Operator

Shortcut for if-else.

- Syntax: condition ? value_if_true : value_if_false
- Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // max = 20
```

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

Decision-Making Statements in C

Decision-making statements allow a program to take different actions based on conditions.

1. if statement

- Executes a block of code if the condition is true.

```
int age = 20;
if (age >= 18) {
printf("You can vote.");
}
```

Output: You can vote.

2. if-else statement

- Executes one block if condition is true, otherwise another block.

```
int num = 5;
if (num % 2 == 0) {
printf("Even");
} else {
   printf("Odd");
}
```

Output: Odd

3. Nested if-else

- An if or else statement inside another if-else.

```
int marks = 75;
if (marks >= 90) {
   printf("Grade A");
} else if (marks >= 60) {
   printf("Grade B");
} else {
   printf("Grade C");
}
```

Output: Grade B

4. switch statement

- Used when you need to choose one option from many cases.

```
int day = 3;
switch(day) {
   case 1: printf("Monday"); break;
   case 2: printf("Tuesday"); break;
   case 3: printf("Wednesday"); break;
   default: printf("Invalid day");
}
```

Output: Wednesday

Summary:

- if → single condition.
- if-else → two possible paths.
- nested if-else → multiple conditions.

- switch → multiple choices (better than many if-else).

6) 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

1. while loop

- **Checks condition first**, then runs code.
- Runs **0 or more times**.

```
int i = 1;

while(i <= 5) {

   printf("%d ", i);

   i++;

}
```

Output: 1 2 3 4 5

Use when number of iterations is *not fixed* (e.g., read input until user enters 0).

2. for loop

- Compact form: initialization, condition, update in one line.
- Runs **0 or more times**.

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

Output: 1 2 3 4 5

Use when number of iterations is *known in advance* (e.g., print 1 to 100).

3. do-while loop

- **Runs code first**, then checks condition.
- Runs **at least once**, even if condition is false.

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

Output: 1 2 3 4 5

Use when code must run *at least once* (e.g., menu-driven programs, asking user input).

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

Jump Statements in C

1. break statement

- Used to **exit from a loop or switch** immediately.
- Control moves outside the loop/switch.

Example:

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

Output: 1 2

2. continue statement

- Skips the **current iteration** and goes to the **next iteration** of the loop.

Example:

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

```
}
```

Output: 1 2 4 5

3. goto statement

- Used to **jump to a labeled statement** in the program.
- Not recommended in modern coding (can make code messy).

Example:

```
#include <stdio.h>
int main() {
    int n = 3;
    if(n == 3) {
        goto label;   // jump to label
    }
    printf("This will be skipped\n");
label:
    printf("Jumped using goto");
    return 0;
}
```

Output: Jumped using goto

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

Functions in C

A **function** in C is a block of code that performs a specific task.

- It helps in **reusing code**, **organizing programs**, and making them **easy to read**.

1. Function Declaration (Prototype)

- Tells the compiler about the function name, return type, and parameters.
- Written before main().

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

2. Function Definition

- Contains the actual code (body) of the function.
- Defines what the function will do.

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

3. Function Call

- Used inside main() or another function to execute it.
- Pass required values (arguments) while calling.

```
int result = add(5, 3);   // function call
```

Example Program

```
#include <stdio.h>

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

int main() {
    int x = 10, y = 20, sum;

    // Function Call
    sum = add(x, y);

    printf("Sum = %d", sum);
    return 0;
}

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

Output:

```
Sum = 30
```

Summary:

- **Declaration** → tells compiler about the function.
- **Definition** → actual code of function.
- **Call** → execute function inside main() or another function.

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

Arrays in C

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

- Helps to store multiple values under one name.
- Accessed using **index** (first element starts at index 0).

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
printf("%d", numbers[2]);  // Output: 30
```

1. One-Dimensional Array (1D)

- A simple list of elements (like a row).
- Syntax: data_type array_name[size];

Example:

```
int marks[3] = {85, 90, 78};
printf("First mark: %d", marks[0]);  // Output: 85
```

2. Multi-Dimensional Array (2D, 3D, etc.)

(a) Two-Dimensional Array (2D)

- Looks like a **table with rows and columns**.
- Syntax: data_type array_name[rows][cols];

Example:

```
int matrix[2][3] = {
```

```
    {1, 2, 3},
    {4, 5, 6}
};
printf("%d", matrix[1][2]);  // Output: 6
```

(b) Three-Dimensional Array (3D)

- Like a **cube (layers of 2D tables)**. Rarely used in simple programs.

Difference Between 1D and Multi-Dimensional Arrays

Feature	1D Array	Multi-Dimensional Array
Structure	Linear list	Table (2D), Cube (3D), etc.
Syntax	int arr[5];	int arr[3][3];
Example Access	arr[2] → 3rd element	arr[1][2] → element at row 2, col 3
Use Case	Store simple lists	Store tables, matrices, grids

Summary:

- Arrays store multiple values of the same type.
- **1D Array** → single row list.
- **2D/3D Array** → tables or cubes.

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

Pointers in C

A **pointer** is a variable that stores the **memory address** of another variable.

- Instead of storing data directly, it stores *where the data is located in memory*.

1. Declaration of Pointers

- Syntax:
- data_type *pointer_name;
- Example:

```
int *p;   // pointer to an integer
char *c;  // pointer to a character
```

2. Initialization of Pointers

- Assign the **address of a variable** using the & (address-of) operator.
- Example:

```
int num = 10;
int *p = &num;  // pointer p stores address of num
```

3. Accessing Data using Pointers

- Use * (dereference operator) to get the value stored at that address.
- Example:

```
printf("Address: %p\n", p);   // prints address
printf("Value: %d\n", *p);   // prints value at that address → 10
```

Why Are Pointers Important in C?

1. **Dynamic Memory Allocation** → allows use of memory during runtime (malloc, free).
2. **Efficiency** → direct access to memory makes C fast.
3. **Array & String Handling** → arrays and strings are closely related to pointers.
4. **Function Arguments** → can pass by reference (useful for swapping values, etc.).
5. **System-Level Programming** → used in OS, drivers, and embedded systems.

Summary:

- A **pointer** stores the address of a variable.
- Declared with *, initialized with &.
- Important for memory management, arrays, strings, and system programming.

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

String Handling Functions in C

(C library <string.h> is required to use them.)

1. strlen() – String Length

- Returns the number of characters in a string (excluding \0).

```
#include <stdio.h>
#include <string.h>
int main() {
   char name[] = "Karan";
   printf("Length = %lu", strlen(name));  // Output: 5
}
```

Useful when checking string size (e.g., validating password length).

2. strcpy() – String Copy

- Copies one string into another.

```
char src[] = "Hello";
char dest[20];
strcpy(dest, src);
printf("%s", dest);   // Output: Hello
```

Useful when duplicating a string.

3. strcat() – String Concatenation

- Joins (appends) one string to the end of another.

```
char s1[20] = "Hello ";
char s2[] = "World";
strcat(s1, s2);
printf("%s", s1);   // Output: Hello World
```

Useful when combining first name + last name, or building messages.

4. strcmp() – String Compare

- Compares two strings:
  - 0 → if equal
  - <0 → if first < second
  - >0 → if first > second

```
char a[] = "apple";
char b[] = "banana";
printf("%d", strcmp(a, b));  // Output: negative value
```

Useful when sorting names or checking login credentials.

5. strchr() – Find Character in String

- Returns pointer to first occurrence of a character in string.

```
char text[] = "Hello";
char *ptr = strchr(text, 'e');
printf("%s", ptr);   // Output: ello
```

Useful when searching a specific character in text (e.g., @ in email).

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

Structures in C

A **structure** in C is a user-defined data type that groups different types of variables under one name.

- Useful when we want to store related data of different types together.
- Example: a **student** has name (string), age (int), marks (float).

1. Declaring a Structure

- Use the keyword struct.

```
struct Student {
   char name[20];
```

```
    int age;
    float marks;
};
```

2. Initializing a Structure

- Create variables of structure type and assign values.

```
struct Student s1 = {"Karan", 20, 85.5};
```

3. Accessing Structure Members

- Use **dot (.) operator** to access members.

```
printf("Name: %s\n", s1.name);
printf("Age: %d\n", s1.age);
printf("Marks: %.2f\n", s1.marks);
```

Example Program

```
#include <stdio.h>
struct Student {
    char name[20];
    int age;
    float marks;
};

int main() {
    struct Student s1 = {"Karan", 20, 85.5};

    printf("Name: %s\n", s1.name);
    printf("Age: %d\n", s1.age);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

Output:

```
Name: Karan
Age: 20
Marks: 85.50
```

13) 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

File handling allows a C program to store data permanently on disk (not just in memory).

- Without file handling → data is lost when program ends.
- With file handling → data can be **saved, modified, and reused**.

Importance of File Handling

1. **Permanent storage** of data.
2. **Large data management** (e.g., databases, records).
3. **Easy access** to read/write files.
4. Used in **real applications** like text editors, banking software, etc.

Basic File Operations in C

1. Opening a File

- Use fopen(filename, mode) → returns a FILE*.
- Modes:
  - "r" → read
  - "w" → write (creates new, erases old)
  - "a" → append (adds at end)
  - "r+", "w+", "a+" → read + write

```
FILE *fp;
fp = fopen("data.txt", "w");  // open for writing
```

2. Closing a File

- Use fclose(fp) to free resources.

```
close(fp);
```

3. Writing to a File

- fprintf(fp, ...) → formatted writing
- fputs(str, fp) → write string

```
FILE *fp = fopen("data.txt", "w");
fprintf(fp, "Hello C Programming\n");
fclose(fp);
```

4. Reading from a File

- fscanf(fp, ...) → formatted reading
- fgets(str, size, fp) → read string

```
FILE *fp = fopen("data.txt", "r");
char line[50];
fgets(line, sizeof(line), fp);
printf("%s", line);
fclose(fp);
```