Module 2 – Introduction to Programming (C Programming)

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

   C programming was developed in 1972 by **Dennis Ritchie** at Bell Laboratories. It was designed as an improvement of the B language to develop the UNIX operating system. C provided features like low-level memory access, efficient execution, and portability, which made it ideal for system software.
   C influenced many modern languages like C++, Java, and Python. It is still widely used today in **embedded systems, operating systems, game engines, and compilers** because of its performance and direct hardware control.
   Thus, C remains a powerful and essential language in the programming world.

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. Download a C compiler:
   - Windows: Install MinGW (includes GCC).
   - Linux/macOS: GCC is usually pre-installed.
2. Add the compiler path to system variables (Windows).
3. Install an IDE such as **DevC++**, **Code::Blocks**, or **VS Code**.
4. Configure the IDE to recognize GCC compiler.
5. Create a new project, write code, compile, and run.

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

- **Header files**: Include predefined functions (e.g., <stdio.h>).
- **main()**: Entry point of a C program.
- **Comments**: Used for documentation (// or /* ... */).
- **Data types**: Define variable type (int, float, char).
- **Variables**: Store values in memory.

Example:

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


int main() {
   int age = 20;       // integer variable
   float marks = 85.5;  // float variable
   char grade = 'A';    // char variable


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


   return 0;
}
```

Output:

```
Age: 20
Marks: 85.50
Grade: A
```

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

- Arithmetic: + - * / % (used in calculations).
- Relational: == != > < >= <= (compare values).
- Logical: && || ! (combine conditions).
- Assignment: = += -= *= /= (assign values).
- Increment/Decrement: ++ -- (increase/decrease).
- Bitwise: & | ^ ~ << >> (work on bits).
- Conditional: ?: (shorthand if-else).

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

- **if**: Executes code when condition is true.
- **else**: Executes when if condition is false.
- **nested if-else**: Multiple conditions tested.
- **switch**: Selects one case out of many.

Example:

```
int num = 2;


if(num > 0)
   printf("Positive\n");
else
   printf("Negative\n");


switch(num) {
```

```
   case 1: printf("One"); break;
   case 2: printf("Two"); break;
   default: printf("Other");
}
```

Output:

```
Positive
Two
```

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

- **while**: Condition checked before loop. Use when iterations unknown.
- **for**: Initialization, condition, increment in one line. Use when iterations known.
- **do-while**: Executes at least once, even if condition is false.

Example:

```
int i = 1;
// while loop
while(i <= 3) {
   printf("%d ", i);
   i++;
}

// for loop
for(int j = 1; j <= 3; j++) {
   printf("\n%d ", j);
}
```

```
// do-while loop
int k = 1;
do {
   printf("\n%d ", k);
   k++;
} while(k <= 3);
```

Output:

```
1 2 3
1 2 3
 1 2 3
```

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

- **break**: Exit loop immediately.
- **continue**: Skip current iteration.
- **goto**: Jump to labeled statement.

Example:

```
for(int i=1; i<=5; i++) {
   if(i==3) continue;   // skip 3
   if(i==5) break;     // stop at 4
   printf("%d ", i);
}


goto label;  // jump
label:
printf(" End");
```

Output:

```
1 2 4 End
```

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

A **function** is a block of reusable code.

- **Declaration**: Defines function prototype.
- **Definition**: Provides code of function.
- **Call**: Executes function.

Example:

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


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


int main() {
   int result = add(5, 3); // call
   printf("Sum = %d", result);
   return 0;
}
```

Output:

```
Sum = 8
```

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

- **Array**: Collection of same type elements stored in continuous memory.
- **1D Array**: Single row of elements.
- **2D Array**: Matrix-like arrangement.

Example:

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


printf("%d\n", arr[1]);       // prints 2
printf("%d\n", matrix[1][0]);  // prints 3
```

Output:

```
2
3
```

🔹 **Difference between One-Dimensional and Multi-Dimensional Arrays**

Aspect	One-Dimensional Array	Multi-Dimensional Array
Definition	Stores elements in a single row (linear form).	Stores elements in rows and columns (tabular form) or higher dimensions.
Declaration	int arr[5];	int arr[3][3]; (2D), int arr[2][3][4]; (3D)
Memory Layout	Elements are stored sequentially in one line.	Elements are stored in row-major order (row by row).
Accessing Elements	Accessed using single index → arr[i].	Accessed using multiple indices → arr[i][j] (2D), arr[i][j][k] (3D).
Use Cases	Used when data is linear (marks of students, temperatures over days).	Used when data is tabular or grid-like (matrices, images, tables).

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

- **Pointer**: A variable storing memory address of another variable.
- **Declaration**: int *ptr;
- **Initialization**: int a=10; int *ptr=&a;
  Importance: Efficient memory use, dynamic allocation, passing by reference, arrays, structures.

Example:

```
int a = 10;

int *p = &a;

printf("Value: %d\n", *p);
```

Output:

```
Value: 10
```

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

- **strlen(str)**: Returns length of string.
- **strcpy(dest, src)**: Copies string.
- **strcat(s1, s2)**: Concatenates strings.
- **strcmp(s1, s2)**: Compares two strings.
- **strchr(str, ch)**: Finds first occurrence of character.

Example:

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

```
char str1[20] = "Hello";
char str2[20] = "World";


printf("%lu\n", strlen(str1)); // 5
strcat(str1, str2);
printf("%s\n", str1); // HelloWorld
```

Output:

```
5
HelloWorld
```

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

A **structure** groups different data types under one name.

- **Declaration**:

```
struct Student {
   char name[20];
   int roll;
   float marks;
};
```

- **Initialization & Access**:

```
struct Student s1 = {"John", 1, 95.5};
printf("%s %d %.1f", s1.name, s1.roll, s1.marks);
```

Output:

```
John 1 95.5
```

13. Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.

- File handling allows **permanent storage** of data.
- Functions:
    - fopen(filename, mode) – open file.
    - fprintf() / fputs() – write.
    - fscanf() / fgets() – read.
    - fclose() – close file.

Example:

```
FILE *f;
f = fopen("test.txt", "w");
fprintf(f, "Hello World");
fclose(f);

f = fopen("test.txt", "r");
char str[50];
fgets(str, 50, f);
printf("%s", str);
fclose(f);
```

Output:

```
Hello World
```