Module 2 – Introduction to Programming

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

➢ **History of C**

- C was developed in the **early 1970s** by **Dennis Ritchie** at **Bell Labs**.
- It came from earlier languages: **BCPL → B → C**.
- In **1973**, the **UNIX operating system** was written in C, making it famous.
- In **1978**, *"The C Programming Language"* book by **Kernighan & Ritchie** was published.
- Later, standards were created: **ANSI C (1989).**

➢ **Evolution**

- C influenced many languages: **C++, Java, Python, C#, Go, Rust**.
- Used in **system programming, compilers, operating systems, and embedded systems**.

➢ **Importance**

- **Fast & Efficient** – close to hardware, very quick.
- **Portable** – programs can run on many machines.
- **Foundation Language** – many modern languages are based on C.
- **System Development** – used in **Linux, Windows, macOS, embedded devices**.
- **Educational Value** – teaches memory, pointers, data structures, algorithms.

➢ **Why Still Used Today?**

- **Speed** (better than most languages).
- **Trusted** (used for decades).
- **Essential in OS, IoT, robotics, embedded systems**.
- **Best for learning basics of programming**.

---

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.

A. Installing GCC Compiler:

➢ The most widely used C compiler is **GCC (GNU Compiler Collection)**.

1. On Windows

- **Step 1:** Download **MinGW** (Minimalist GNU for Windows) from https://www.mingw-w64.org.
- **Step 2:** Run installer → select **C Compiler (gcc)** during installation.
- **Step 3:** Add the `bin` folder path (e.g., `C:\MinGW\bin`) to **System Environment Variables → PATH**.
- **Step 4:** Open **Command Prompt** → type:
  gcc --version

If it shows version info, GCC is installed successfully ✓.

B. Setting up IDEs

➢ **Visual Studio Code (VS Code)**

- Download VS Code from https://code.visualstudio.com.
- Install **C/C++ extension** from Microsoft.
- Install GCC (via MinGW on Windows, or directly on Linux/Mac as above).
- Configure **tasks.json** (for build) and **launch.json** (for debugging).
- Save program with .c extension → press **Ctrl+Shift+B** to build → run in terminal.

---

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 generally has these parts:

- **Header Files (Preprocessor Directives)**
  - Written at the top using `#include`
  - They tell the compiler to include libraries (like `stdio.h` for input/output).

Example:

```
#inlcude<stdio.h>
```

- **Comments**
  - Used to explain code.
  - Compiler ignores them.
  - Two types:

1.Single line:

```
//THIS IS SINGLE LINE COMMENT.
```

2. Multi Line:

```
/* THIS IS MULTILINE
COMMENT IN
C PROGRAMMING */
```

- **Main Function**

- Execution of every C program starts from main().
- Syntax:

```
int main() {
        //code
        return 0;
}
```

- **Data Types:**

- Tell the type of data a variable can store.
- Common types:
  - int → integers (10, -5)
  - float → decimal numbers (3.14, -2.5)
  - char → single characters ('A', 'b')
  - double → large decimal numbers
- Example:

```
#include<stdio.h>
void main()
{
        int age; //This is an integer datatype
        age=10;
}
```

- **Variables**

  - Named storage locations in memory.
  - Must be declared with a data type before use.

Example:

```
//Print different data types

void main()

{

        int age=10;

        char name='s';

        float percent=90.35;

        printf("Age is %d",age);

        printf("\nFirst letter of name is %c",name);

        printf("\nPercentage is %f ",percent);

}
```

Output:

```
Age is 10

First letter of name is s

Percentage is 90.35
```

---

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

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

- <u>Arithmetic Operators</u>

- Used for mathematical calculations.

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

- Relational Operators

  - Used to compare two values. Result is **true (1)** or **false (0)**.

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

- Logical Operators

  - Used for decision-making (combine conditions).

Operator	Meaning	Example
&&	Logical AND (true if both true)	(a > 5 && b < 10)
||	Logical OR(true if either true)	(a>5 || b<10)
!	Logical NOT (reverses result)	!(a > 5)

- Assignment Operators

  - Used to assign values to variables.

Operator	Meaning	Example
=	Assigns value	a = 5
+=	Add and assign	a += 3; // a = a + 3
-=	Subtract and assign	a -= 2; // a = a - 2
*=	Multiply and assign	a *= 2; // a = a * 2
/=	Divide and assign	a /= 2; // a = a / 2
%=	Modulus and assign	a %= 3; // a = a % 3

- Increment & Decrement Operators

➢ Used to increase or decrease value of a variable by 1.

Operator	Meaning	Example
++a	Pre-increment (increase before use)	If a=5, ++a → 6
a++	Post-increment (increase after use)	If a=5, a++ → 5 then 6
--a	Pre-decrement (decrease before use)	If a=5, --a → 4
a--	Post-decrement (decrease after use)	If a=5, a-- → 5 then 4

- Bitwise Operators

➢ Work at the **bit-level** (0s and 1s).

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

- Conditional (Ternary) Operator in C

- It is a **shortcut for if-else**.
- Uses **three operands**, so it is called *ternary*.

  - Syntax: (condition) ? expression1 : expression2;

    - If condition is true → expression1 executes
    - If condition is false → expression2 executes

---

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

- In **C programming**, decision-making statements are used when you want your program to take different actions depending on conditions (true/false).

1. if Statement

- Executes a block of code only **if the condition is true**.
- If condition is false, the block is skipped.
- **Syntax:**

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

- Example:

```
#include<stdio.h>
void main()
{
   int age=20;
   if(age>=18)
   {
      printf("You are eligible for vote");
   }
  printf("\n End of program");
```

```
}
```

2. if-else Statement

- Executes one block **if condition is true**, otherwise executes another block.
- Syntax:

```
If(condition)
{
//Code executes when condition is true
}
else
{
//code executes when condition is false
}
```

- Example:

```
#include<stdio.h>
void main()
{
int age=10;
if(age>=18)
{
printf("You are eligible for vote");
}
Else
{
printf("You are not eligible for vote");
}
}
```

3. Nested if-else

- An **if-else inside another if-else**.
- Used when multiple conditions are checked step by step.
- Syntax:

```
if (condition1)
{
   // code if condition1 is true
}
else if (condition2)
{
   // code if condition2 is true
}
 else
{
   // code if none are true
}
```

Example:

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

```
void main()
{
int  marks=60;
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");
}
```

```
}
```

4. switch Statement

- Used when you need to **choose from multiple options** based on a single expression.
- Each case represents a possible value.
- Syntax:

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

Example:

```
#include <stdio.h>
int main() {
   int choice = 2;
   switch (choice)
 {
      case 1:
         printf("You selected Option 1.\n");
         break;
      case 2:
         printf("You selected Option 2.\n");
         break;
      case 3:
         printf("You selected Option 3.\n");
         break;
      default:
```

```
        printf("Invalid choice.\n");
    }
    return 0;
}
```

---

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

1. While Loop

- **Definition**: An entry-controlled loop (condition is checked before execution).
- **Syntax**:

```
while (condition)
{
   // statements
}
```

- **Key Point**: The loop body executes **only if the condition is true at the start**.
- **Best for**: When the number of iterations is **unknown** in advance (e.g., reading input until the user enters 0).

2. For Loop

- **Definition**: An entry-controlled loop with initialization, condition, and increment/decrement all in one line.
- **Syntax**:

```
for (initialization; condition; update)
 {
   // statements
}
```

- **Key Point**: Provides a compact structure, ideal when the **number of iterations is known**.
- **Best for**: Counting loops, like printing numbers from 1 to 100, iterating over arrays, etc.

3. Do-While Loop

- **Definition**: An exit-controlled loop (condition is checked **after** execution).
- **Syntax**:

```
do {
   // statements
} while (condition);
```

- **Key Point**: The body executes **at least once**, even if the condition is false initially.
- **Best for**: When you need the loop body to execute at least once (e.g., showing a menu to the user before checking if they want to continue).

Feature	while loop	for loop	do-while loop
Type	Entry-controlled	Entry-controlled	Exit-controlled
Condition check	Before loop body	Before loop body	After loop body
Minimum execution	0 times	0 times	At least 1 time
Use case	Unknown iterations	Known iterations	Must run at least once
Compactness	Moderate (separate initialization)	High (all in one line)	Moderate (condition at end)

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

1. break Statement

- **Use**: Immediately exits from the **nearest loop** (for, while, do-while) or from a switch statement.
- **Effect**: Control jumps to the first statement after the loop/switch.
- Example**:**

```
#include<stdio.h>

void main()
```

```
{
int i;
for(i=1;i<=5;i++)
{
if(i==4)
{
break;
}
printf("%d\n",i);
}
}
```

2. continue Statement

- **Use**: Skips the current iteration of a loop and moves to the **next iteration**.
- **Effect**: Condition is checked again for the next cycle.
- **Example**:

```
#include<stdio.h>
void main()
{
int i;
For(i=1;i<=5;i++){
If(i==3)
Continue;
Printf("%d\n",i);
} }
```

3. goto Statement

- **Use**: Transfers control to another part of the program using a **label**.
- **Effect**: Can move control forward or backward.
- **Example**:

```
#include<stdio.h>

int main() {

   int i = 1;

   start:                          // label

   printf("%d ", i);

   i++;

   if (i <= 5) {

      goto start;                  // jumps back to label

   }

   return 0;

}
```

---

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

- A **function** is a block of code that performs a specific task.
- Functions help in **code reusability, modularity, readability, and debugging**.
- Types of Functions:

1. **Library Functions** → Already defined in header files.
   - Example: printf(), scanf().
2. **User-defined Functions** → Created by the programmer to perform specific tasks.

- A function in C has **three important parts**:

1. Function Declaration (Prototype)

- Tells the compiler about the function's **name, return type, and parameters** (if any).

- Syntax:

```
return_type  function_name(parameter_list);
```

Example:

```
int add(int, int);
```

2. Function Definition

- Actual body of the function, where the task is performed.
- Syntax:

```
return_type function_name(parameter_list) {
   // statements
   return value;   // if return_type is not void
}
```

Example:

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

3. Function Call

- Used to **execute the function** in the main program.
- Syntax:

```
function_name(arguments);
```

Example:

```
int result = add(5, 3)
```

Example:

#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;
}
```

---

9. 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 under a single variable name and accessing them using an **index**.
- Key points:

- Index starts from **0**.
- Elements are stored sequentially in memory.

I. <u>One-Dimensional (1D) Array</u>

- A **1D array** is like a simple list of elements in a single row.
- <u>Syntax:</u>

  data_type array_name[size];

- <u>Example:</u>

```
#include <stdio.h>

int main() {

   int marks[5] = {90, 85, 70, 95, 88};      // 1D Array

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

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

   }

   return 0;

}
```

II. Multi-Dimensional Array

- A **multi-dimensional array** is an array of arrays.
- Syntax: data_type array_name[rows][columns];
- Example:

```
#include <stdio.h>

int main() {

   int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};    // 2x3 matrix

   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;

}
```

🔻 Difference Between 1D and Multi-Dimensional Arrays

Feature	1D Array	Multi-Dimensional Array (e.g., 2D)
Structure	Linear (single row)	Tabular (rows × columns)
Syntax	int arr[5];	int arr[3][4];
Accessing elements	arr[i]	arr[i][j]
Storage	List of elements	Matrix / table / higher dimensions
Example use	Storing marks of 5 students	Storing marks in 3 subjects for 5 students

---

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

- A **pointer** is a special variable in C that stores the **address** of another variable. Instead of holding a value directly, it "points" to the location in memory where the value is stored.

- Declaration of Pointers
  Syntax:
  data_type *pointer_name;

- data_type → type of data the pointer will point to.
- * → indicates that it's a pointer.

Example:

```
int *ptr;     // pointer to an integer
char *cptr;   // pointer to a character
float *fptr;  // pointer to a float
```

- Initialization of Pointers
  - A pointer is initialized by storing the **address of a variable** using the **address-of operator (&)**.

Example:

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

```
int main() {
   int x = 10;
   int *p;      // pointer declaration
   p = &x;      // initialization (p stores address of x)

   printf("Value of x = %d\n", x);          //It will print 10 as x value
   printf("Address of x = %p\n", &x);  //It will printf address
   printf("Pointer p stores address = %p\n", p); //It also print address
   printf("Value pointed by p = %d\n", *p);  // dereferencing
   return 0;
}
```

- Why Are Pointers Important in C?

➢ Pointers are **powerful** and give C its efficiency. They are important because:

1. **Dynamic Memory Allocation**
   - With pointers, you can allocate memory at runtime using malloc(), calloc(), free().
2. **Efficient Array & String Handling**
   - Arrays and strings can be processed faster using pointers instead of indexing.
3. **Function Arguments (Call by Reference)**
   - Pointers allow functions to modify actual variables, not just copies.
4. **Data Structures**
   - Used to create linked lists, trees, graphs, and other dynamic structures.
5. **Memory Management & Low-Level Access**
   - Pointers allow direct interaction with memory, making C close to hardware.

---

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

All these functions are defined in the **<string.h>** library.

1. strlen() – String Length

- **Purpose:** Returns the length of a string (number of characters before \0).
- **Syntax:**

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

Example:

```
#include <stdio.h>

#include <string.h>

int main() {

   char name[] = "Shifa";

   printf("Length of string = %d\n", strlen(name));

   return 0;

}
```

2. strcpy() – String Copy

- **Purpose:** Copies one string into another.
- **Syntax:** char strcpy(char *destination, const char *source);
- **Example:**

```
#include <stdio.h>

#include <string.h>

int main() {

   char src[] = "Hello";

   char dest[20];

   strcpy(dest, src);

   printf("Copied string: %s\n", dest);

   return 0;

}
```

3. strcat() – String Concatenation

- **Purpose:** Appends (joins) one string to the end of another.
- **Syntax: char strcat(char *destination, const char *source);**
- **Example:**

```
#include <stdio.h>

#include <string.h>

int main() {

   char s1[30] = "Hello ";

   char s2[] = "World!";

   strcat(s1, s2);

   printf("Concatenated string: %s\n", s1);

   return 0;

}
```

4. strcmp() – String Compare

- **Purpose:** Compares two strings.
- **Syntax: int strcmp(const char *str1, const char *str2);**

Returns:

- 0 → if strings are equal.
- <0 → if str1 < str2.
- >0 → if str1 > str2.

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("'%s' comes before '%s'\n", a, b);
    else
        printf("'%s' comes after '%s'\n", a, b);
    return 0;
}
```

5. strchr() – String Character Search

- **Purpose:** Finds the first occurrence of a character in a string.
- **Syntax:** char* strchr(const char *str, int character);
- Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "programming";
    char *pos = strchr(str, 'g');
    if (pos != NULL)
        printf("First occurrence of 'g' is at index %ld\n", pos - str);
    else
        printf("Character not found\n");
    return 0;
}
```

---

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

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

❖ Declaring a Structure

- You use the keyword struct to define a structure.

```
struct Student
 {
   int  roll_no;       // integer member
   char name[50];      // character array (string)
   float marks;        // float member
};
```

- Here:

- struct Student is the **structure definition**.
- It does not create a variable, it just defines a template (like a blueprint).

❖ Declaring Structure Variables

- You can declare variables in two ways:

1. Separately after defining

   struct Student s1, s2;

2. Together with structure definition

   struct Student {

      int roll_no;

      char name[50];

      float marks;

   } s1, s2;

❖ Initializing a Structure

- You can initialize members when declaring:

struct Student s1 = {1, "Shifa", 89.5};

❖ Accessing Structure Members

- Use the **dot operator (.)** with the structure variable.

```
printf("Roll No: %d\n", s1.roll_no);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);
```

- Example Program:

```
#include <stdio.h>
#include <string.h>
// Structure definition
struct Student {
   int roll_no;
   char name[50];
   float marks;
};
int main() {
   // Initialization during declaration
   struct Student s1 = {1, "Shifa", 89.5};
   // Assigning values later
   struct Student s2;
   s2.roll_no = 2;
   strcpy(s2.name, "Arman");
   s2.marks = 92.0;
   // Accessing members
   printf("Student 1: %d, %s, %.2f\n", s1.roll_no, s1.name, s1.marks);
```

```
    printf("Student 2: %d, %s, %.2f\n", s2.roll_no, s2.name, s2.marks);

    return 0;

}
```

---

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

- In C programming, **file handling** allows us to store data permanently on storage devices like hard disks instead of losing it when the program ends (as happens with variables stored in RAM).

- **Importance**

1. **Permanent storage** – Unlike variables, files keep data even after program termination.
2. **Large data management** – Files can store huge amounts of data efficiently.
3. **Data sharing** – Files make it possible to share information between programs.
4. **Reusability** – Data saved once can be read and reused multiple times.
5. **Security** – Files can be password-protected or encrypted.

- File Operations in C:
- To work with files, C provides functions in the **stdio.h** library.

- **Basic File Operations**

1. **Opening a file** – Using fopen().
2. **Closing a file** – Using fclose().
3. **Reading from a file** – Using fscanf(), fgets(), fgetc().
4. **Writing to a file** – Using fprintf(), fputs(), fputc().

- File Handling Functions

1. Opening a File

```
FILE *fp;

fp = fopen("example.txt", "w");   // Open file in write mode
```

i. "r" → read
ii. "w" → write (creates new file or overwrites existing)
iii. "a" → append (adds data at end)

2. Closing a File:

```
fclose(fp);   // Always close to save changes & free memory
```

3. Writing to a File:

```
#include <stdio.h>
int main() {
   FILE *fp;
   fp = fopen("example.txt", "w");
   if (fp == NULL) {
      printf("Error opening file!\n");
      return 1;
   }
   fprintf(fp, "Hello, File Handling in C!\n");
   fputs("This is a sample line.\n", fp);
   fclose(fp);
   return 0;
}
```

4. Reading from a File:

```
#include <stdio.h>
int main() {
   FILE *fp;
   char str[100];
   fp = fopen("example.txt", "r");
   if (fp == NULL)
 {
```

```
        printf("Error opening file!\n");
            return 1;
        }
        while (fgets(str, sizeof(str), fp) != NULL) {
            printf("%s", str);   // Print line by line
        }
        fclose(fp);
        return 0;
    }
```
