Module 2 – Introduction to Programming

THEORY EXERCISE

1. Overview of C Programming

- **C programming is a powerful, general-purpose programming language developed in the early 1970s by Dennis Ritchie at Bell Labs. It is known for its speed, efficiency, and control over system resources, making it widely used in system/software development.**

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

    1) **Origins: From B to a Systems Language**

Birth of C: In the early 1970s, Dennis Ritchie at Bell Labs created C (1972), evolving from Ken Thompson's language B (itself inspired by BCPL) to better support UNIX development

Rewrite of UNIX: By 1973, UNIX was largely reimplemented in C, greatly enhancing its portability across hardware platforms

2) **Popularization & Standardization**

- **K&R & the C book (1978): Brian Kernighan and Dennis Ritchie co-authored *The C Programming Language*—the de facto reference known as "K&R C," nurturing widespread academic and industry adoption**
- **ANSI/ISO standards:**
  - **C89/C90 (ANSI C): Introduced in 1989–90 with function prototypes and richer standard libraries**
  - **C99 (1999): Brought modern features like inline functions, long long, single-line comments, and variable-length arrays.**
  - **C11 (2011): Added threads, Unicode support, atomics, and type-generic macros .**
  - **C17/C18 (2018): Primarily a bug-fix release .**

- **C23 (2024): The current standard, introducing features like nullptr and binary literals.**

3) Evolution: Legacy and Offshoots

- **Rise of C-derived languages: C influenced C++ (1985), which in turn shaped Java, C#, and later Rust and Go.**
- **Intermediate role: C often serves as a portable intermediate for compilers and tools—even acting as a backend or a metastructure language.**

4) Why C Still Matters Today

1. **Unmatched Performance & Low-level Access**
    - **Compiled code runs close to hardware; fine control over memory, pointers, and CPU operations is essential for OS kernels, device drivers, embedded firmware, and high-frequency systems.**
    - **Its efficiency underpins scientific computing and high-performance libraries (e.g. NumPy, SciPy)**
2. **Exceptional Portability**
    - **"Portable assembly"—C code compiles on virtually any platform with minimal changes thanks to a consistent standard library.**
3. **Deep Influence & Interoperability**
    - **Foundational feedstock for modern languages; essential for interfacing via C foreign function interfaces (e.g. Python's C extensions) .**
4. **Vast Legacy Codebase**
    - **Key OS (Linux, Windows, macOS), databases (MySQL, PostgreSQL, SQLite), compilers (GCC, CPython), web servers (Apache, Nginx)—all heavily written in C, requiring expertise for ongoing maintenance .**
5. **Essential Teaching Tool**
    - **Teaches core computer science concepts: memory management, data structures, pointers, and system organization—offering unparalleled insight into how computers work.**

5) Endurance into the Future

- **Embedded/IoT domination: C remains the premier language for microcontrollers, automotive controllers, IoT firmware, real-time systems, and industrial devices worldwide.**
- **Kernel & system-critical roles: OS kernels and BIOS/firmware continue to depend on C—no higher-level alternative achieves the same fine-grained control.**
- **Stable foundation: Mature platforms like Linux, Git, and the LAMP stack highlight how software "matures" over time—open-source C software is foundational, reliable, and long-lived.**
- **Standards evolve: Despite being over half a century old, C evolves via C23 and future standards—showing a living language adapting to modern hardware and developer needs.**

6) Conclusion

C began as a solution to make UNIX portable and efficient. Through its concise design and powerful features, it became the bedrock for modern computing.

Its performance, portability, transparency, and ubiquitous ecosystem have secured its place in programming history—and its adaptability guarantees C remains relevant, teaching generations and powering critical systems for years to come.

2. Setting Up Environmen:

❖ **Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like DevC++, VS Code, or Code Blocks.**

1) Install Dev-C++ (bundled with GCC)

1. **Download Dev-C++ (Embarcadero version) from SourceForge. It includes a MinGW-based GCC compiler.**
2. **Run the installer, accept defaults, and ensure the option to install the MinGW compiler suite is checked .**
3. **When prompted, allow Dev-C++ to set up its compiler paths automatically.**

2) Verify and Configure

- **Launch Dev-C++ (C:\Dev-Cpp\devcpp.exe by default).**

- It auto-detects GCC/Mingw. To check: go to Tools → Compiler Options, and verify paths point to MinGW64\bin or MinGW32\bin.

3) Write, Compile & Run

- Create a New Project → Console Application → C language in Dev-C++.
- Write your .c code.
- Press F9 (Compile & Run) or click the Run ▶ button.
- If everything's configured correctly, a console window displays the output.

4) (Optional) Add Compiler to Windows PATH

- Find the bin folder under Dev-C++ install (e.g., C:\Dev-Cpp\MinGW64\bin).
- Add it to the System PATH via Environment Variables.
- Open new CMD and run gcc --version — it should work.

➢ Install Code::Blocks with GCC:

Code :: Blocks is another beginner-friendly IDE that can include GCC.

Steps:

1. Go to: https://www.codeblocks.org/downloads/
2. Download the "codeblocks-XXmingw-setup.exe" version *(includes GCC compiler)*.
3. Run the installer and select default options.
4. Open Code::Blocks.
5. Go to File → New → Project → Console Application → C.
6. Write your code and press F9 to build and run.

➢ Install VS Code with GCC (Advanced Setup):

Visual Studio Code (VS Code) is a modern, powerful code editor. You need to manually install GCC and some extensions.

Steps:

1. Install GCC Compiler (via MinGW):

- Go to: https://www.mingw-w64.org/
- Download and install the version for Windows.
- During installation:
    - Architecture: x86_64
    - Threads: posix
    - Exception: seh
- Add the bin folder (e.g., C:\Program Files\mingw-w64\...\bin) to your System PATH:
    - Right-click This PC → Properties → Advanced System Settings → Environment Variables → PATH → Edit and paste the path.

2. Verify GCC Installation:

- Open Command Prompt and type:

css

gcc --version

If installed correctly, it will show the version.

3. Install VS Code:

- Download from: https://code.visualstudio.com/
- Install and launch it.

4. Install C/C++ Extension:

- Go to Extensions (Ctrl+Shift+X), search for C/C++, and install Microsoft's C/C++ extension.

5. Create and Run a C File:

- Create a new folder and open it in VS Code.
- Create a file program.c, write your code.
- Open the terminal in VS Code (Ctrl + ~) and compile:

bash

CopyEdit

```
gcc program.c -o program

./program
```

Summary:

IDE	Compiler Needed	Difficulty	Best For
DevC++	Built-in (GCC)	Easy	Beginners
Code::Blocks	Built-in (with MinGW)	Easy	Beginners to Intermediate
VS Code	Requires GCC setup	Medium	Intermediate to Advanced

3. Basic Structure of a C Program

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

1) Headers / Preprocessor

- Include standard libraries with #include, letting you use built-in functions:

```
c

#include <stdio.h>   // printf, scanf

#include <stdlib.h>  // malloc, free
```

- You can also define constants or macros:

```
c

#define PI 3.14
```

2) Comments

- Single-line: // ...
- Multi-line: /* ... */
- Used to explain code—ignored during compilation:

```
c

// compute the sum

/*

 This function does more

 complex stuff...

*/
```

3. main() Function

- Entry point of every C program:

```
c

int main(void) {

   // code goes here

   return 0;  // signals success

}
```

4. Data Types & Variables

- Basic types:
    - int, char, float, double, _Bool
- Declare variables before use:

```
c

int count = 5;
```

```
float average;

char letter = 'A';

_Bool done = 0;
```

5. Putting It All Together

```
c

#include <stdio.h>



#define MAX 10  // max items



int main(void) {

   // variable declarations

   int i;

   float sum = 0.0f;



   for (i = 0; i < MAX; i++) {

      sum += i;  // accumulate

   }



   printf("Sum = %.2f\n", sum);

   return 0;
```

```
}
```

4. Operators in C

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

X1. Arithmetic Operators

These perform basic mathematical operations.

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

Example:

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

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

2. Relational Operators

These compare two values and return true (1) or false (0).

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b

Operator	Description	Example
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

Example:

```
c

if (a > b) {

   printf("a is greater than b");

}
```

3. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example
&&	Logical AND (both true)	(a > 5 && b < 10)
`		`
!	Logical NOT (negation)	!(a > 5)

4. Assignment Operators

Assign values to variables.

Operator	Description	Example
=	Assign value	a = 5
+=	Add and assign	a += 2 → a = a + 2
-=	Subtract and assign	a -= 2
*=	Multiply and assign	a *= 2

Operator	Description	Example
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 2

5. Increment and Decrement Operators

Used to increase or decrease a value by 1.

Operator	Description	Example
++	Increment	a++ or ++a
--	Decrement	a-- or --a

- a++ → Post-increment (use a, then increment)
- ++a → Pre-increment (increment, then use a)

6. Conditional (Ternary) Operator

A shorthand for if-else statement.

Syntax:

c

(condition) ? expression1 : expression2;

Example:

```
c

int max = (a > b) ? a : b;
```

5.Control Flow Statements in C.

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

1. if Statement

It checks a condition. If true, it executes the block.

```
int num = 10;

if (num > 0) {

   printf("Positive number\n");

}
```

1. if-else Statement

It provides two paths: one if the condition is true, and another if it's false.

```
int num = -5;

if (num >= 0) {

   printf("Non-negative number\n");

} else {

   printf("Negative number\n");

}
```

Nested if-else Statement

You can put one if or if-else inside another. Used for multiple conditions

```
int num = 0;

if (num >= 0) {

   if (num == 0) {

      printf("Zero\n");
```

```
    } else {
        printf("Positive number\n");
    }
} else {
    printf("Negative number\n");
}
```

switch Statement

It is used to select one block of code from multiple options based on the value of a variable.

```
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("Other day\n");

}
```

Statement Type	Use Case Example
If	Single condition
if-else	Two conditions (true/false)
Nested if-else	Multiple conditions/levels
Switch	Multiple constant value checks

6. Looping in C

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

1) while Loop

Description:

- **Checks the condition first.**
- **Executes the loop only if the condition is true.**
- **It's a pre-test loop.**

```
int i = 1;

while (i <= 5) {

   printf("%d ", i);

   i++;

}
```

2) For loop

Description:

- All loop control (start, end, update) is in one line.
- Also a pre-test loop (condition is checked first).
- Most compact loop.

Example:

```
c

for (int i = 1; i <= 5; i++) {

   printf("%d ", i);

}
```

Best Use:

- When the number of iterations is known.
- Example: Printing the first 10 natural numbers.

3) do-while Loop

Description:

- Executes the loop at least once, then checks the condition.
- It's a post-test loop.

Example:

```
c

int i = 1;

do {

   printf("%d ", i);

   i++;
```

```
} while (i <= 5);
```

Best Use:

- **When the loop must run at least once.**
- **Example: Menu-driven programs or user confirmation prompts.**

7. Loop Control Statements

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

break Statement in C

Use:

- **Immediately exits from the loop or a switch block.**
- **Control moves to the statement after the loop/switch.**

Example:

```
c

#include <stdio.h>



int main() {

   for (int i = 1; i <= 5; i++) {

      if (i == 3)

         break;  // Exit loop when i == 3

      printf("%d ", i);

   }

   return 0;
```

```
}
```

Output:

```
1 2
```

continue Statement in C

Use:

- Skips the current iteration of the loop.
- Loop continues with the next iteration.Example:

```
c



#include <stdio.h>

int main() {

   for (int i = 1; i <= 5; i++) {

      if (i == 3)

         continue;  // Skip printing when i == 3

      printf("%d ", i);

   }

   return 0;

}
```

Output:

```
1 2 4 5
```

goto Statement in C

Use:

- **Transfers control to a labeled statement.**
- **Avoid using unless absolutely necessary (makes code harder to read/debug).**

Example:

```
c

#include <stdio.h>



int main() {

   int i = 1;

   loop:

   if (i <= 3) {

      printf("%d ", i);

      i++;

      goto loop;  // Jump back to the label 'loop'

   }

   return 0;

}
```

Output: 1 2 3

8.Functions in C

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

In **C programming**, a **function** is a block of code that performs a **specific task**. Functions make code **modular, reusable, and easier to manage**.

3 Main Parts of a Function

1.Function Declaration **(also called *function prototype*)**

- Tells the compiler about the **function name, return type, and parameters**.

2. Function Definition

- Contains the **actual code** for the function.

3. Function Call

- Invokes or **executes the function**.

```
c
#include <stdio.h>

// 1. Function declaration
int multiply(int, int);

int main() {
   // 2. Function call
   int result = multiply(4, 5);
   printf("Product: %d\n", result);
   return 0;
}

// 3. Function definition
int multiply(int a, int b) {
   return a * b;
}
```

9. Arrays in C

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

In C, an **array** is a fixed-size, contiguous collection of elements of the same type, accessed by index. Here's a concise breakdown:

1) One-Dimensional Array

c

```c
#include <stdio.h>


int main(void) {
   int arr[5] = {1, 2, 3, 4, 5};  // declares an array of 5 ints
   arr[2] = 10;                // modify the 3rd element

   for (int i = 0; i < 5; i++)
      printf("%d ", arr[i]);    // prints: 1 2 10 4 5

   return 0;
}
```

Explanation:

- int arr[5] reserves five contiguous int slots.
- arr[2] accesses/modifies the third element (index starts from 0).
- A for loop iterates over all elements

2) Two-Dimensional Array

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


int main(void) {
```

```
    int mat[2][3] = { {1, 2, 3}, {4, 5, 6} };  // 2 rows × 3 columns


    mat[1][2] = 9;  // set element at row 2, column 3


    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
    // prints:
    // 1 2 3
    // 4 5 9


    return 0;
}
```

Explanation:

- **int mat[2][3] defines a 2×3 matrix (stored row-major).**
- **mat[i][j] accesses the element at row i and column j.**
- **Nested loops print each row.**

Comparison

Feature	**1D Array**	**Multi-Dimensional Array**
Syntax	type name[size];	type name[rows][cols];
Access	a[i]	a[i][j]
Use case	Lists, sequences	Matrices, tables, grids

Feature	1D Array	Multi-Dimensional Array
Memory ordering	Linear	Contiguous blocks, row-first (row-major)

Example Code:

```
c

CopyEdit

#include <stdio.h>


int main(void) {

   int nums[5] = {1, 2, 3, 4, 5};

   int mat[2][3] = { {1,2,3}, {4,5,6} };


   printf("%d, %d\n", nums[2], mat[1][1]);  // prints: 3, 5


   return 0;

}
```

Arrays in C provide efficient and direct access to collections of data.

10. Pointers in C

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

- Pointers in C are Variables that store the store the memory address of another variables. They are declared using the "*" symbol, and you can access the value stored at the memory address using the dereference operator "*" Pointers allow direct memory manipulation, making them powerful for dynamic memory allocation and working with arrays, function, and structure. The address of a variable is obtained using the address- of operator '&'.
- A **pointer** is a variable that **holds the memory address** of another variable, not its actual value.

1) Declared

Use * to declare a pointer to a specific data type:

```
c

int *p;     // p is a pointer to an int
```

```
char *cptr;  // cptr is a pointer to a char
```

---

2) Initialization

Pointers must be given an address—either of a variable or NULL:

a) Point to a variable:

```
c
int x = 10;
int *p = &x;  // &x gives the address of x
```

b) Assign later:

```
c
int *p;
p = &x;      // now p points to x
```

c) Initialize as NULL (safer default):

```
c
int *p = NULL;
```

Summary Table

Feature	**Syntax**	**Purpose**
Declare a pointer	int *p;	Define that p holds an int address
Initialize pointer	p = &x;	Store address of x in p
Dereference pointer	*p	Access or modify the value at address
Pass by reference	func(int *p)	Allow function to alter caller's data
Dynamic memory alloc.	malloc()	Work with heap memory

11. Strings in C

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

- A **string** is a **sequence of characters** stored in **an array** and ends with a special character: '\0' (called null character).

1. strlen() – String Length

- **Purpose:**

Returns the **number of characters** in a string (excluding the '\0' null character).

- **Syntax:**

```
c
int length = strlen(string);
```

- **Example:**

```
c
char name[] = "Hello";
int len = strlen(name);   // len = 5
```

- **Use Case:**

Useful when you want to **count characters**, like in password validation or data limits.

2. strcpy() – Copy One String to Another

- **Purpose:**

Copies the **source string** into the **destination string**.

- **Syntax:**

```
c
strcpy(destination, source);
```

- **Example:**

```
c
char src[] = "India";
char dest[20];
strcpy(dest, src);   // dest now contains "India"
```

- **Use Case:**

Used to **duplicate a string**, or transfer value from one string to another.

3. strcat() – String Concatenation (Joining)

- **Purpose:**

Appends (adds) one string at the end of another.

- **Syntax:**

```
c
strcat(string1, string2);  // string2 is added to string1
```

- **Example:**

```
c
char s1[20] = "Good ";
char s2[] = "Morning";
strcat(s1, s2);   // s1 becomes "Good Morning"
```

- **Use Case:**

Useful when making **full sentences**, file paths, or joining inputs.

4. strcmp() – String Comparison

- **Purpose:**

Compares two strings **character by character**.

- **Syntax:**

```
c
int result = strcmp(str1, str2);
```

- **Return Values:**
  - 0 → strings are **equal**
  - <0 → str1 is **less** than str2
  - >0 → str1 is **greater** than str2
- **Example:**

```
c
strcmp("abc", "abc") → 0
strcmp("abc", "abd") → -1
strcmp("abd", "abc") → 1
```

- **Use Case:**

Used for **checking if two strings match**, like login username/password, search, etc.

5. strchr() – Find First Occurrence of Character

- **Purpose:**

Searches for the **first occurrence** of a character in a string.

- **Syntax:**

```
c
char *ptr = strchr(str, 'a');
```

- **Example:**

```
c

char str[] = "Gopal";

char *p = strchr(str, 'p');  // returns pointer to 'p'

printf("Character found at: %s", p);  // Output: "pal"
```

Use Case:

Useful in **searching specific letters**, email format validation (@), file extensions (.), etc.

12. Structures in C

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

1. What is a Struct?

A struct in C is a composite data type that lets you group variables of different types under one name.

2. Declaring a Struct Type

```
c

struct Point {

   int x;

   int y;

};
```

This defines a new type struct Point with members x and y

3. Creating Variables

After defining, you can create variables:

```
c

struct Point p1, p2;
```

Or combine declaration with definition:

```
c

struct Point { int x, y; } p3;
```

4. Initializing Structs

- **Positional (C89 style)**

```
c
CopyEdit
struct Point p = {3, 4};
```

- **Designated (C99+)**

```
c
struct Point p = {.y = 4, .x = 3};
```

Unspecified members default to zero

5. Accessing Members

- With a struct variable:

```
c
CopyEdit
p.x = 10;
printf("%d\n", p.y);
```

- With a pointer to a struct:

```
c
struct Point *pp = &p;
pp->y = 20;
```

6. Copying Structs

You can assign structs directly:

```
c
struct Point p2 = p1;   // Copy all members
```

13. File Handling in C

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 (Theory Answer)

🔷 **Introduction:**

In C language, **file handling** is used to store data permanently in a file.
It allows reading from and writing to files on the disk, so data is not lost when the program ends.

🔷 **Importance of File Handling:**

1. **Permanent Storage:** Data remains saved even after the program ends.
2. **Large Data Handling:** We can read/write large amounts of data using files.
3. **Reuse of Data:** Once written, data can be reused many times.
4. **Real Applications:** Used in record systems, billing software, database programs, etc.

🔷 **File Operations in C:**

Operation	Function Used	Description
Open a file	fopen()	Opens a file in a specific mode (read/write)
Write to file	fprintf() / fputs()	Writes data to file
Read from file	fscanf(), fgetc(), fgets()	Reads data from file
Close file	fclose()	Closes the opened file

🔷 **File Opening Modes in fopen():**

Mode Meaning

"r" Read only

"w" Write only (erases old data)

"a" Append (add data at end)

"r+" Read and Write

"w+" Write and Read (erases old data)

"a+" Append and Read

🔷 **Example Code (Basic Idea):**

```
c
```

```
FILE *fp;
fp = fopen("data.txt", "w");  // Open file to write
fprintf(fp, "Hello World!");  // Write data
fclose(fp);                   // Close file
```