Module 2 : Introduction to Programming

Overview of C Programming

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

→ The History and Evolution of C Programming

The C programming language stands as one of the most influential technologies in modern computing. Its development fundamentally shaped operating systems, systems programming, and even the design of many contemporary programming languages. Understanding C's history provides valuable insight into how computing evolved from early machine-specific languages to today's complex, high-level ecosystems.

Origins in the 1960s and Early 1970s

C traces its roots to the late 1960s at Bell Labs. Before C, programmers relied heavily on assembly languages, which were fast and efficient but highly machine-dependent. In an effort to create a higher-level yet powerful language, researchers at Bell Labs experimented with various languages, including **BCPL** (Basic Combined Programming Language) and **B**. These predecessors offered some portability but lacked features needed for large-scale systems programming.

In 1972, **Dennis Ritchie** developed C as an evolution of B, adding data types, structures, and a more robust syntax. The breakthrough came when C was used to rewrite the Unix operating system, which had originally been implemented in assembly. This was revolutionary—no major operating system had ever been written in a high-level language before. Because C was efficient and near the hardware, Unix maintained its speed while gaining portability, allowing it to spread rapidly across diverse computer systems.

Standardization and Expansion

As C grew more widely used, variations began to appear. To ensure consistency, the **American National Standards Institute (ANSI)** standardized the language in 1989, creating **ANSI C** (also known as **C89** or **C90**). This standard cemented C's

syntax, library functions, and behavior across compilers, making it reliable for commercial and academic use.

Subsequent updates—such as **C99**, **C11**, **C17**, and **C23**—introduced features like inline functions, improved memory handling, multi-threading support, and safer libraries. These revisions balanced new capabilities with the language's core philosophy: simplicity, low-level control, and efficiency.

C's Influence on Other Languages

C's impact extends far beyond its own ecosystem. Many modern languages—including **C++**, **Java**, **C#**, **Objective-C**, **Rust**, and **Go**—inherit syntax, principles, or memory models from C. This "C-family" lineage means that learning C provides a foundation that directly supports understanding a large portion of today's programming landscape.

Why C Is Important

1. Close-to-Hardware Control

C offers direct manipulation of memory through pointers and supports low-level operations unavailable in many high-level languages. This makes it ideal for:

- Operating system kernels
- Embedded systems
- Device drivers
- Real-time applications

Programs written in C can be extremely fast and predictable, critical for systems where timing and resource use matter.

2. Portability

C was designed with portability in mind. Programs can be compiled on different hardware architectures with minimal changes, which contributed greatly to Unix's spread and remains a major advantage today.

3. Efficiency and Performance

C compilers produce highly optimized machine code. For performance-critical applications—such as game engines, databases, and cryptographic tools—C remains one of the best choices.

4. Foundation for Modern Computing

Because major operating systems and tools are written in C or rely heavily on it, understanding C helps developers understand the underlying mechanisms of:

- Linux
- Windows
- MacOS
- Embedded firmware
- Compilers and interpreters

Why C Is Still Used Today

1. Ubiquity in Systems Programming

Much of modern infrastructure—from servers to IoT devices—runs code written in C. Rewriting these systems in new languages would be costly and risky, so C remains entrenched.

2. Longevity and Stability

The language evolves cautiously. This stability reassures engineers who require long-term support for industrial, medical, or aerospace systems where reliability is essential.

3. Interoperability

C serves as the “common language” among programming environments. Libraries written in C can easily be used by languages like Python, Java, or Rust, making C a backbone of cross-language development.

4. Active Community and Tooling

Decades of use have produced robust compilers, debuggers, static analyzers, and support communities. Even as new languages arise, C’s ecosystem remains unmatched in maturity.

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

→ 1. Embedded Systems (Microcontrollers, IoT Devices, Automotive Electronics)

C is the dominant language in embedded programming because it provides direct control over hardware, low memory usage, and real-time performance.

Examples of real-world use:

- Automotive control units (ECUs): Engine control, braking systems (ABS), airbag controllers, and infotainment units rely heavily on C.
- Consumer electronics: TVs, microwaves, routers, smart thermostats, and wearables use C-based firmware.
- Industrial and medical devices: Robotics controllers, pacemakers, and MRI machines use C for deterministic, reliable performance.

2. Operating Systems (Linux, Windows, macOS)

C was originally created to rewrite the Unix operating system—setting a precedent that continues today. Most modern OS kernels and low-level system utilities are still written predominantly in C.

Examples of real-world use:

- Linux kernel: ~80% written in C
- Microsoft Windows: Core kernel and driver layers heavily use C
- macOS & iOS (XNU kernel): Large sections implemented in C
- Android: Many system libraries and components rely on C and C++ through the NDK
- Predictable performance and precise control over hardware resources

3. Game Development (Game Engines, Performance-Critical Systems)

While high-level languages and scripts often handle gameplay logic, the core engines that power games still use C or C++ due to performance demands.

Examples of real-world use:

- Unity Engine: Core runtime components use C and C++
- Unreal Engine: Built with C and C++, with C-based modules for physics and rendering
- Nintendo and PlayStation SDKs: Provide C/C++ APIs for hardware-level game development
- Custom graphics and physics engines: Often implemented in C for speed and efficiency.

2. Setting Up Environment

- **THEORY EXERCISE:**
  - 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. Installing a C Compiler (GCC)

GCC (GNU Compiler Collection) is one of the most widely used compilers for the C programming language.

A. Installing GCC on Windows (Using MinGW-w64)

Steps:

1. **Download MinGW-w64**
   - Search online for **"MinGW-w64 download"**.
   - Open the official MinGW-w64 download page and download the installer.
2. **Run the Installer**
   - During installation, select the following options:
     - **Architecture:** x86_64 (for 64-bit Windows)
     - **Threads:** posix
     - **Exception:** seh
3. **Choose Installation Directory**
   - Select an installation path, for example:
   - C:\mingw-w64
4. **Add GCC to System PATH**
   - Open **Control Panel → System → Advanced system settings**
   - Click **Environment Variables**
   - Under **System Variables**, select **Path** and click **Edit**
   - Add the following path:

- C:\mingw-w64\bin

5. **Verify Installation**
    - Open **Command Prompt**
    - Type the following command:
    - gcc --version
    - If GCC is installed correctly, the version information will be displayed.

2. Setting Up an IDE for C Programming

An IDE (Integrated Development Environment) helps you write, compile, and run C programs easily. Below are setup steps for three popular IDEs: **Dev-C++**, **Visual Studio Code**, and **CodeBlocks**.

Option 1: Setting Up Dev-C++

Dev-C++ is beginner-friendly and includes a built-in MinGW compiler.

Steps:

1. Search online for **"Dev-C++ download"** (Orwell or Embarcadero edition).
2. Download and install Dev-C++ using the installer.
3. Launch Dev-C++.
4. Create a new source file:
    - Go to **File → New → Source File**
5. Write your C program and save the file with a **.c** extension.
6. Click **Compile & Run** to execute the program.

Option 2: Setting Up Visual Studio Code (VS Code)

VS Code is a lightweight and powerful editor. GCC must be installed separately.

Steps:

1. Download and install **Visual Studio Code**.
2. Open VS Code and go to **Extensions** (left sidebar).
3. Install the following extensions:
    - **C/C++** (by Microsoft)
    - **Code Runner** (optional)
4. Ensure **GCC is installed** and added to the system PATH.
5. Create a new folder for your C project and open it in VS Code.
6. Create a file named main.c.
7. Open the terminal and compile the program:
8. gcc main.c -o main
9. Run the compiled program:
10. ./main
11. For one-click compilation, press **Ctrl + Shift + B** and let VS Code generate tasks.json.

Option 3: Setting Up CodeBlocks

CodeBlocks is an easy-to-use IDE and can be installed with a built-in compiler.

Steps:

1. Search for **"CodeBlocks download"**.
2. Download the version labeled:
   CodeBlocks with MinGW (includes GCC compiler).
3. Install CodeBlocks using default settings.
4. Open CodeBlocks and create a new project:
    - **File → New → Project → Console Application → C**
5. Enter the project name and choose a folder.
6. CodeBlocks will generate a sample main.c file.
7. Press **Build & Run (F9)** to compile and execute the program.

- **LAB EXERCISE:**
  - Install a C compiler on your system and configure the IDE. Write your first program to print "Hello, World!" and run it.

→ **Install a C Compiler (GCC)**

Option A: Windows (Using MinGW-w64)

1. Search: **“MinGW-w64 download”**
2. Download the installer (choose the SourceForge or official link).
3. During setup:
   - Architecture: **x86_64**
   - Threads: **posix**
4. Install to:
   C:\mingw-w64
5. Add it to PATH:
   - Control Panel → System → Advanced System Settings
   - Environment Variables → Path → Add:
     C:\mingw-w64\bin
6. Verify:
   Open Command Prompt and type:

   gcc --version

   PROGRAM:

   #include <stdio.h>

   int main() {

   printf("Hello, World!\n");

   return 0;

   }

   OUTPUT:

   Hello, World!

3. Basic Structure of a C Program

- **THEORY EXERCISE:**
  - 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

C program generally consists of the following parts:

1. Header Files
2. Main Function
3. Declarations (Data Types and Variables)
4. Statements
5. Comments

1. Header Files

Header files contain predefined functions and library definitions that are used in a C program.

Example:

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

- stdio.h provides input and output functions such as printf() and scanf().

2. The main() Function

Every C program must have a main() function.
Program execution always starts from the main() function.

Example:

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

- return 0; indicates that the program has executed successfully.

3. Comments

Comments are explanatory notes written for programmers.
They are ignored by the compiler and do not affect program execution.

Types of Comments:

Single-line comment

```
// This is a single-line comment
```

Multi-line comment

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

4. Data Types

Data types specify the type of data that a variable can store.

Data Type	Meaning	Example Value
int	Integer numbers	10, -5
float	Decimal numbers	3.14
double	Large decimal numbers	12.98765
char	Single character	'A'
char[]	String (character array)	"Hello"

5. Variables

A variable is used to store data in memory.
Variables must be declared before they are used.

Declaration Examples:

```
int age;

float price;

char letter;
```

Declaration with Initialization:

```
int age = 20;
float price = 5.99;
char letter = 'A';
```

Putting It All Together: Complete C Program Example

```
#include <stdio.h>   // Header file
int main() {        // Main function

   // Declaring variables
   int age = 18;
   float height = 5.6;
   char grade = 'A';

   // Printing output
   printf("Age: %d\n", age);
   printf("Height: %.1f\n", height);
   printf("Grade: %c\n", grade);

   return 0;       // Program ends
}
```

Output:

```
Age: 18
Height: 5.6
Grade: A
```

- **LAB EXERCISE:**
  - Write a C program that includes variables, constants, and comments. Declare and use different data types (int, char, float) and display their values.

→ **PROGRAM:**

```
#include <stdio.h>
int main() {



  int age = 20;
  float height = 5.9;
  char grade = 'A';
  const int BIRTH_YEAR = 2003;


  printf("\n Age: %d ", age);
  printf("\n Height: %.1f ", height);
  printf("\n Grade: %c ", grade);
  printf("\n Birth Year (constant): %d ", BIRTH_YEAR);
  return 0;  // program ends
}
```

OUTPUT:

Age: 20

Height: 5.9

Grade: A

Birth Year (constant): 2003

4. Operators in C

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

→ **1. Arithmetic Operators**

Used for **mathematical calculations**.

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

2. Relational Operators

Used to **compare values**. Returns **1 (true)** or **0 (false)**.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

3. Logical Operators

Used to **combine conditions**. Works with true (1) and false (0).

Operator	Description	Example
&&	Logical AND	a && b
!	Logical NOT	!a

4. Assignment Operators

Used to **assign values** to variables. Can also combine operations.

Operator	Description	Example
=	Assign	a = 10
+=	Add and assign	a += 5 (a = a + 5)
-=	Subtract and assign	a -= 3
*=	Multiply and assign	a *= 2
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 3

5. Increment and Decrement Operators

Used to **increase or decrease** a variable by 1.

Operator	Description	Example
++	Increment by 1	a++ or ++a
--	Decrement by 1	a-- or --a

Note: ++a is **pre-increment** (increment before use), a++ is **post-increment** (increment after use).

6. Bitwise Operators

Used for **bit-level operations**.

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

7. Conditional (Ternary) Operator

A **shortcut for if-else**.

Syntax:

condition ? value_if_true : value_if_false;

- **LAB EXERCISE:**
  - Write a C program that accepts two integers from the user and performs arithmetic, relational, and logical operations on them. Display the results.

→ **PROGRAM:**

```
#include <stdio.h>

int main() {

int a, b;


printf("Enter first integer: ");

scanf("%d", &a);


printf("Enter second integer: ");

scanf("%d", &b);


printf("\n--- Arithmetic Operations ---\n");

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

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

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


if (b != 0) {

   printf("Division (a / b) = %d\n", a / b);

   printf("Modulus (a %% b) = %d\n", a % b);

} else {

   printf("Division and Modulus not possible (division by zero)\n");

}
```

```
    printf("\n--- Relational Operations ---\n");
    printf("a == b : %d\n", a == b);
    printf("a != b : %d\n", a != b);
    printf("a > b  : %d\n", a > b);
    printf("a < b  : %d\n", a < b);
    printf("a >= b : %d\n", a >= b);
    printf("a <= b : %d\n", a <= b);


    printf("\n--- Logical Operations ---\n");
    printf("(a && b) : %d\n", a && b);
    printf("(a || b) : %d\n", a || b);
    printf("!(a)     : %d\n", !a);
    printf("!(b)     : %d\n", !b);


}
```

OUTPUT:

Enter first integer: 10

Enter second integer: 5

--- Arithmetic Operations ---

Addition (a + b) = 15

Subtraction (a - b) = 5

Multiplication (a * b) = 50

Division (a / b) = 2

Modulus (a % b) = 0

--- Relational Operations ---

a == b : 0

a != b : 1

a > b : 1

a < b : 0

a >= b : 1

a <= b : 0

--- Logical Operations ---

(a && b) : 1

(a || b) : 1

!(a) : 0

!(b) : 0

5. Control Flow Statements in C

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

→

1️⃣ if Statement

🔹 **Definition**

The if statement executes a block of code **only if the condition is true**.

🔹 **Syntax**

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

🔹 **Example**

```
#include <stdio.h>
int main() {
int num = 10;

if (num > 0) {
printf("Number is positive");
}

return 0;
}
```

2 if-else Statement

- **Definition**

  The if-else statement executes:

  - if block when condition is true
  - else block when condition is false

- Syntax

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

- Example

```
#include <stdio.h>

int main() {
 int num = -5;

if (num >= 0) {
printf("Number is non-negative");
} else {
printf("Number is negative");
 }

 return 0;
}
```

3 Nested if-else Statement

◆ Definition

When one if-else statement is placed **inside another**, it is called nested if-else. Used for **multiple conditions**.

◆ Syntax

```
if (condition1) {
   if (condition2) {
      // code
   } else {
      // code
   }
} else {
   // code
}
```

◆ Example

```
#include <stdio.h>
int main() {
   int marks = 75;
   if (marks >= 40) {
      if (marks >= 75) {
         printf("Grade: Distinction");
      } else {
         printf("Grade: Pass");
      }
   } else {
      printf("Grade: Fail");
   }
}
```

4️⃣ switch Statement

- **Definition**

  The switch statement selects one block of code to execute from **multiple choices**, based on the value of an expression.

- **Syntax**

```
switch (expression) {
   case value1:
      // code
      break;
   case value2:
      // code
      break;
   default:
      // code
}
```

- **Example**

```
#include <stdio.h>

int main() {
   int day = 3;

   switch (day) {
      case 1:
         printf("Monday");
         break;
```

```
        case 2:
            printf("Tuesday");
            break;
        case 3:
            printf("Wednesday");
            break;
        case 4:
            printf("Thursday");
            break;
        case 5:
            printf("Friday");
            break;
        default:
            printf("Invalid day");
    }

    return 0;
}
```

- **LAB EXERCISE:**
  - Write a C program to check if a number is even or odd using an if-else statement. Extend the program using a switch statement to display the month name based on the user's input (1 for January, 2 for February, etc.).

→ **PROGRAM:**

```
#include <stdio.h>

int main() {
   int num, month;

   printf("Enter a number: ");
   scanf("%d", &num);

   if (num % 2 == 0) {
      printf("The number %d is Even\n", num);
   } else {
      printf("The number %d is Odd\n", num);
   }

   // Month name using switch
   printf("\nEnter month number (1-12): ");
   scanf("%d", &month);
```

```
switch (month) {
  case 1:
     printf("January");
     break;
  case 2:
     printf("February");
     break;
  case 3:
     printf("March");
     break;
  case 4:
     printf("April");
     break;
  case 5:
     printf("May");
     break;
  case 6:
     printf("June");
     break;
  case 7:
     printf("July");
     break;
  case 8:
     printf("August");
     break;
```

```
        case 9:
            printf("September");
            break;
        case 10:
            printf("October");
            break;
        case 11:
            printf("November");
            break;
        case 12:
            printf("December");
            break;
        default:
            printf("Invalid month number");
    }

    return 0;
}
```

6. Looping in C

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

→

1️⃣ while Loop

🔹 **Description**

- Entry-controlled loop
- Condition is checked **before** executing the loop body
- Loop may execute **zero times**

🔹 **Syntax**

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

🔹 **Example**

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

2️⃣ for Loop

🔹 **Description**

- Entry-controlled loop
- Initialization, condition, and increment/decrement in **one line**
- Loop may execute **zero times**

🔹 **Syntax**

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

🔹 **Example**

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

3️⃣ do-while Loop

🔹 **Description**

- Exit-controlled loop
- Condition is checked after executing the loop body
- Loop executes at least once

🔹 **Syntax**

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

🔹 **Example**

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

🔍 Comparison Table

Feature	while	for	do-while
Condition check	Before loop	Before loop	After loop
Executes at least once	❌ No	❌ No	✅ Yes
Best for	Unknown iterations	Known iterations	Menu-driven programs
Loop control	Separate	Compact	Separate
Readability	Medium	High	Medium

- **LAB EXERCISE:**
  - Write a C program to print numbers from 1 to 10 using all three types of loops (while, for, do-while).

→ **PROGRAM:**

```
#include <stdio.h>


int main() {
   int i;



   printf("Using while loop:\n");
   i = 1;
   while (i <= 10) {
      printf("%d ", i);
      i++;
   }
```

```
    printf("\n\nUsing for loop:\n");
    for (i = 1; i <= 10; i++) {
        printf("%d ", i);
    }


    printf("\n\nUsing do-while loop:\n");
    i = 1;
    do {
        printf("%d ", i);
        i++;
    } while (i <= 10);

    return 0;
}
```

OUTPUT:

```
Using while loop:
1 2 3 4 5 6 7 8 9 10

Using for loop:
1 2 3 4 5 6 7 8 9 10

Using do-while loop:
1 2 3 4 5 6 7 8 9 10
```

7. Loop Control Statements

- **THEORY EXERCISE:**
  - Explain the use of break, continue, and goto statements in C. Provide examples of each.

→ In C, **break**, **continue**, and **goto** are **jump statements** used to alter the normal flow of program execution. Below is a clear explanation with **examples** and **use cases**.

1️⃣ break Statement

🔹 Use

- Immediately **terminates a loop** (for, while, do-while) or a switch statement
- Control moves to the statement **after** the loop or switch

🔹 Example (using loop)

```
#include <stdio.h>


int main() {
   int i;


   for (i = 1; i <= 10; i++) {
      if (i == 5) {
         break;
      }
      printf("%d ", i);
   }
}
```

OUTPUT:

```
1 2 3 4
```

2️⃣ continue Statement

- **Use**
  - Skips the **current iteration** of a loop
  - Control jumps to the **next iteration** of the loop
- **Example**

```
#include <stdio.h>

int main() {
    int i;

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

    return 0;
}
```

OUTPUT:

```
1 2 4 5
```

3 goto Statement

Use

- Transfers control to a **labeled statement** within the same function
- Generally **discouraged** because it makes code hard to read and debug

Syntax

```
goto label;
/* ... */
label:
    statement;
```

Example

```
#include <stdio.h>
int main() {
    int num;

    printf("Enter a positive number: ");
    scanf("%d", &num);

    if (num < 0) {
        goto error;
    }

    printf("You entered: %d", num);
    return 0;

error:
    printf("Error: Negative number entered");
    return 0;
}
```

- **LAB EXERCISE:**
  - Write a C program that uses the break statement to stop printing numbers when it reaches 5. Modify the program to skip printing the number 3 using the continue statement.

→ **PROGRAM:**

```
#include <stdio.h>
int main() {
   int i;
   printf("Using break statement:\n");
   for (i = 1; i <= 10; i++) {
      if (i == 5) {
         break;
      }
      printf("%d ", i);
   }

   printf("\n\nUsing continue statement:\n");
   for (i = 1; i <= 10; i++) {
      if (i == 3) {
         continue;
      }
      printf("%d ", i);
   }
}
```

OUTPUT:

Using break statement:

1 2 3 4

Using continue statement:

1 2 4 5 6 7 8 9 10

8. Loop Control Statements

- **THEORY EXERCISE:**
  - What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

→ A function in C is a self-contained block of code that performs a specific task and is executed when it is called. It helps in **modular programming**, **code reusability**, and **better program organization**.

1️⃣ Function Declaration (Function Prototype)

🔹 **Meaning**

- Tells the compiler **about the function's name, return type, and parameters**
- Written **before main()**
- Does **not** contain the function body

🔹 **Syntax**

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

🔹 **Example**

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

2️⃣ Function Definition

🔹 **Meaning**

- Contains the **actual code** of the function
- Defines **what the function does**
- Written **after main()** (or before)

🔹 **Syntax**

```
return_type function_name(parameter_list) {
   // function body
}
```

🔹 **Example**

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

3️⃣ Function Call

🔹 **Meaning**

- Used to **execute the function**
- Control is transferred to the function, and after execution, returns back

🔹 **Syntax**

```
function_name(arguments);
```

🔹 **Example**

```
sum = add(10, 20);
```

✅ Complete Program Example

```
#include <stdio.h>

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

int main() {
    int result;

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

```
    printf("Sum = %d", result);

    return 0;
}

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

OUTPUT:

```
Sum = 8
```

- **LAB EXERCISE:**
  - Write a C program that calculates the factorial of a number using a function. Include function declaration, definition, and call.

→ **PROGRAM:**

```
#include <stdio.h>

int factorial(int);

int main() {
    int num, result;

    // Input from user
    printf("Enter a number: ");
```

```
    scanf("%d", &num);
    result = factorial(num);

    printf("Factorial of %d = %d", num, result);

    return 0;
}

int factorial(int n) {
    int fact = 1, i;

    for (i = 1; i <= n; i++) {
        fact = fact * i;
    }

    return fact;
}
```

OUTPUT:

```
Enter a number: 5
Factorial of 5 = 120
```

9. Arrays in C

- **THEORY EXERCISE:**
  - 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 and accessed using a common name with an index.

Arrays allow storing **multiple values** in a single variable, making programs more efficient and organized.

1️⃣ One-Dimensional Array (1D Array)

🔹 Definition

A **one-dimensional array** stores elements in a **single row or list**.

🔹 Syntax

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

🔹 Example

```
#include <stdio.h>


int main() {
   int arr[5] = {10, 20, 30, 40, 50};


   for (int i = 0; i < 5; i++) {
      printf("%d ", arr[i]);
   }
}
```

OUTPUT:

```
10 20 30 40 50
```

2️⃣ Multi-Dimensional Array

🔹 Definition

A **multi-dimensional array** is an array with **more than one dimension**, commonly used as a **table or matrix**.

🔹 Two-Dimensional Array (Most Common)

🔹 Syntax

```
data_type array_name[rows][columns];
```

🔹 Example

```
#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 Between 1D and Multi-Dimensional Arrays

Feature	One-Dimensional Array	Multi-Dimensional Array
Structure	Linear list	Table / matrix
Index	One index	Two or more indices
Example	arr[5]	arr[2][3]
Usage	List of marks	Matrix, tables

- **LAB EXERCISE:**
  - Write a C program that stores 5 integers in a one-dimensional array and prints them. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.

→ **PROGRAM:**

```
#include <stdio.h>


int main() {
   int arr[5];
   int matrix[3][3];
   int i, j, sum = 0;



   printf("Enter 5 integers:\n");
   for (i = 0; i < 5; i++) {
      scanf("%d", &arr[i]);
   }
```

```
    printf("Elements of the 1D array:\n");
    for (i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }


    printf("\n\nEnter elements of 3x3 matrix:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &matrix[i][j]);
            sum += matrix[i][j];
        }
    }

    printf("\nMatrix elements:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    printf("\nSum of all matrix elements = %d", sum);
}
```

INPUT:

Enter 5 integers:

1 2 3 4 5

Enter elements of 3x3 matrix:

1 2 3

4 5 6

7 8 9

OUTPUT:

Elements of the 1D array:

1 2 3 4 5

Matrix elements:

1 2 3

4 5 6

7 8 9

Sum of all matrix elements = 45

10. Pointers in C

- **THEORY EXERCISE:**
  - Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?

→ A pointer in C is a variable that stores the memory address of another variable.

Instead of holding a value directly, a pointer **points to the location** where the value is stored.

- **Why Are Pointers Important in C?**

Pointers are important because they:

- Allow **direct memory access**
- Enable **call by reference** in functions
- Help in **dynamic memory allocation**
- Improve **program efficiency**
- Are essential for **arrays, strings, structures, and linked data structures**

- **Declaration of Pointers**
- **Syntax**

```
data_type *pointer_name;
```

- **Example**

```
int *p;
```

- **Initialization of Pointers**
- **Syntax**

```
pointer_name = &variable_name;
```

- **Example**

```
int x = 10;
int *p = &x;
```

- **Accessing Value Using Pointer (Dereferencing)**

```
printf("%d", *p);   // prints value of x
```

- &x → address of variable
- *p → value at that address

PROGRAM:

```
#include <stdio.h>

int main() {
   int x = 10;
   int *p;

   p = &x;

   printf("Value of x = %d\n", x);
   printf("Address of x = %p\n", &x);
   printf("Value stored in pointer p = %p\n", p);
   printf("Value pointed by p = %d\n", *p);

   return 0;
}
```

- **LAB EXERCISE:**
  - Write a C program that stores 5 integers in a one-dimensional array and prints them. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.

→ **PROGRAM:**

```
#include <stdio.h>


int main() {
   int num = 10;      // Normal variable
   int *ptr;          // Pointer declaration


   ptr = &num;        // Pointer initialization


   // Modify value using pointer
   *ptr = 20;


   // Print result
   printf("Value of num = %d\n", num);


   return 0;
}
```

OUTPUT:

```
Value of num = 20
```

11. Strings in C

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

→ In C, **string handling functions** are provided by the header file **<string.h>**. These functions make it easy to **measure, copy, join, compare, and search strings**.

- **1. strlen()**
- **Purpose**

  Finds the length of a string (excluding the null character '\0').
- **Syntax**

  strlen(string);
- **Example**

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

int main() {
   char str[] = "Hello";
   printf("Length = %lu", strlen(str));
   return 0;
}
```

- **Output**

  Length = 5
- **When Useful**
  - Validating input length (passwords, usernames)
  - Looping through characters in a string

- **2. strcpy()**
- **Purpose**

  Copies one string into another.
- **Syntax**

  strcpy(destination, source);
- **Example**

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

int main() {
   char src[] = "C Programming";
   char dest[20];

   strcpy(dest, src);
   printf("Copied string: %s", dest);

   return 0;
}
```

- **Output**

  Copied string: C Programming
- **When Useful**
  - Copying user input
  - Assigning one string to another

- **3. strcat()**
- **Purpose**

  Joins (concatenates) two strings.
- **Syntax**

  strcat(destination, source);
- **Example**

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

int main() {
   char str1[20] = "Hello ";
   char str2[] = "World";

   strcat(str1, str2);
   printf("Combined string: %s", str1);

   return 0;
}
```

- **Output**

  Combined string: Hello World
- **When Useful**
  - Creating full names
  - Joining messages or sentences

- **4. strcmp()**
- **Purpose**

  Compares two strings.
- **Syntax**

  strcmp(string1, string2);
- **Return Values**
  - 0 → strings are equal
  - < 0 → string1 < string2
  - > 0 → string1 > string2
- **Example**

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

int main() {
   char s1[] = "apple";
   char s2[] = "apple";

   if (strcmp(s1, s2) == 0)
      printf("Strings are equal");
   else
      printf("Strings are not equal");
}
```

- **Output**

  Strings are equal
- **When Useful**
  - Login validation
  - Matching passwords or commands

- **5. strchr()**
- **Purpose**

  Finds the first occurrence of a character in a string.
- **Syntax**

  strchr(string, character);
- **Example**

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

int main() {
   char str[] = "Computer";

   char *pos = strchr(str, 'p');

   if (pos != NULL)
      printf("Character found at position: %ld", pos - str);
   else
      printf("Character not found");

   return 0;
}
```

- **Output**

  Character found at position: 2
- **When Useful**
  - Searching characters in text
  - Parsing strings

- **LAB EXERCISE:**
  - Write a C program that takes two strings from the user and concatenates them using strcat(). Display the concatenated string and its length using strlen().

→ **PROGRAM:**

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[50], str2[50];

   // Input strings
   printf("Enter first string: ");
   gets(str1);

   printf("Enter second string: ");
   gets(str2);

   // Concatenate strings
   strcat(str1, str2);

   // Display result
   printf("\nConcatenated string: %s", str1);
   printf("\nLength of concatenated string: %lu", strlen(str1));
}
```

INPUT:

Enter first string: Hello

Enter second string: World

OUTPUT:

Concatenated string: Hello World

Length of concatenated string: 11

12. Structures in C

- **THEORY EXERCISE:**
  - Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

→

🔹 **Concept / Definition**

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

Structures are used to represent **real-world entities** (like a student, employee, book, etc.) in a program.

🔹 **Why Use Structures?**

- Store **different types of data** together
- Organize complex data clearly
- Useful in records, databases, and file handling

1️⃣ **Declaration of a Structure**

🔹 **Syntax**

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
};
```

🔹 **Example**

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

2️⃣ Declaring Structure Variables

```
struct Student s1;
```

Or together with structure definition:

```
struct Student {
   int roll;
   char name[20];
   float marks;
} s1;
```

3️⃣ Initialization of Structure

- **Method 1: At Declaration**

```
struct Student s1 = {1, "Rahul", 85.5};
```

4️⃣ Accessing Structure Members

- **Dot (.) Operator**

Used to access structure members.

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

PROGRAM:

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

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

int main() {
    struct Student s1;

    s1.roll = 101;
    strcpy(s1.name, "Aman");
    s1.marks = 88.5;

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

    return 0;
}
```

- **LAB EXERCISE:**
  - Write a C program that defines a structure to store a student's details (name, roll number, and marks). Use an array of structures to store details of 3 students and print them.

→ **PROGRAM:**

```
#include <stdio.h>

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

int main() {
   struct Student s[3];
   int i;

   // Input student details
   for (i = 0; i < 3; i++) {
      printf("\nEnter details of Student %d\n", i + 1);

      printf("Roll Number: ");
      scanf("%d", &s[i].roll);

      printf("Name: ");
      scanf(" %[^\n]", s[i].name);   // reads full name
```

```
        printf("Marks: ");
        scanf("%f", &s[i].marks);
    }

    // Display student details
    printf("\n--- Student Details ---\n");
    for (i = 0; i < 3; i++) {
        printf("\nStudent %d\n", i + 1);
        printf("Roll Number: %d\n", s[i].roll);
        printf("Name: %s\n", s[i].name);
        printf("Marks: %.2f\n", s[i].marks);
    }

    return 0;
}
```

13. File Handling in C

- **THEORY EXERCISE:**
  - 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 allows a program to store data permanently in files and retrieve it later.
Unlike variables (which store data temporarily in RAM), files store data on **secondary storage** (hard disk).

🔹 **Why File Handling Is Important**

File handling is important because it:

- Provides **permanent storage** of data
- Helps in **data sharing** between programs
- Handles **large amounts of data** efficiently
- Maintains **records** (students, employees, transactions)
- Supports **data backup and recovery**

1️⃣ **Opening a File**

🔹 **Function**

```
FILE *fp;

fp = fopen("file_name", "mode");
```

- **Common File Modes**

Mode	Purpose
"r"	Read
"w"	Write
"a"	Append
"r+"	Read & write
"w+"	Write & read

- **Example**

```
FILE *fp;
fp = fopen("data.txt", "w");
```

2 Writing to a File

- **Functions Used**
  - fprintf() → write formatted data
  - fputc() → write a character
  - fputs() → write a string
- **Example**

```
fprintf(fp, "Hello File");
```

3 Reading from a File

- **Functions Used**
  - fscanf() → read formatted data
  - fgetc() → read a character
  - fgets() → read a string

- **Example**

```
char str[20];
fgets(str, 20, fp);
```

4️⃣ Closing a File

- **Function**

```
fclose(fp);
```

PROGRAM:

```
#include <stdio.h>
int main() {
   FILE *fp;
   char text[50];

   // Writing to file
   fp = fopen("sample.txt", "w");
   fprintf(fp, "Welcome to File Handling in C");
   fclose(fp);

   // Reading from file
   fp = fopen("sample.txt", "r");
   fgets(text, 50, fp);
   printf("File content: %s", text);
   fclose(fp);

   }
```

- **LAB EXERCISE:**
  - Write a C program to create a file, write a string into it, close the file, then open the file again to read and display its contents.

→ **PROGRAM:**

```
#include <stdio.h>


int main() {
   FILE *fp;
   char text[100];


   // Create and write to file
   fp = fopen("myfile.txt", "w");
   if (fp == NULL) {
      printf("File could not be created.");
      return 0;
   }


   printf("Enter a string to write into file: ");
   fgets(text, 100, stdin);


   fputs(text, fp);
   fclose(fp);


   // Open file for reading
   fp = fopen("myfile.txt", "r");
   if (fp == NULL) {
```

```
        printf("File could not be opened.");
        return 0;
    }

    printf("\nFile contents:\n");
    while (fgets(text, 100, fp) != NULL) {
        printf("%s", text);
    }

    fclose(fp);

    return 0;
}
```

INPUT:

Enter a string to write into file:

Welcome to File Handling in C

OUTPUT:

File contents:

Welcome to File Handling in C