1. 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 Origins of C :

The origins of C date back to the late **1960s** when computer science was undergoing a significant transformation. In **1969**, Ken Thompson, a computer scientist at AT&T Bell Labs, created the B programming language as part of the development of the Unix operating system. While B was an improvement over earlier assembly languages, it lacked certain features that made it versatile for larger software projects. Thompson, along with Dennis Ritchie, another researcher at Bell Labs, began developing a new language that would provide more control over hardware while being more portable than assembly language.

In 1972, Ritchie and Thompson released the first version of the C programming language, initially as a way to rewrite the Unix operating system.

C was designed to offer a balance between high-level abstractions and low-level hardware access, a quality that would make it highly adaptable for different systems. C’s syntax was influenced by both B and the earlier language BCPL, and it introduced new features such as data types, structures, and functions, setting it apart from previous programming languages.

The Importance of C

1. Foundation of Operating Systems – The Unix operating system, Linux, Windows components, and embedded systems rely heavily on C.
2. Portability and Efficiency – C code can be compiled on almost any platform, making it ideal for cross-platform applications.
3. Close-to-Hardware Programming – C allows direct manipulation of memory and hardware, making it essential for embedded systems and performance-critical applications.
4. Influence on Other Languages – Many modern programming languages, including C++, Java, Python, and Go, have inherited syntax and concepts from C.

- syntax and concepts from C.

5. Used in Embedded and IoT Systems – Due to its low-level capabilities, C is widely used in microcontrollers, firmware, and Internet of Things (IoT) devices.

Why C Is Still Used Today :

Despite the rise of higher-level languages, C remains relevant because:

- Speed and Performance – C programs execute faster than those written in interpreted languages like Python or JavaScript.
- Control Over System Resources – Unlike many modern languages that abstract hardware details, C gives programmers full control over memory and processor operations.
- Minimal Runtime Overhead – C does not require heavy runtime environments, making it suitable for real-time applications.

Educational Value – Learning C helps programmers understand fundamental programming concepts, such as memory management, pointers, and data structures.

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.

Three Real-World Applications of C Programming.:

1. Operating Systems Development

Example: Linux, Windows Kernel, MacOS

- C is the backbone of most modern operating systems. The Linux kernel and a significant portion of Windows are written in C.
- The macOS and iOS kernels (based on Unix) also use C as their primary programming language.
- C provides direct hardware interaction and memory management, making it ideal for OS development.

2. Embedded Systems & IoT Devices

Example: Smart TVs, Automotive Control Units, Medical Devices

- Embedded systems, such as automobile ECUs (Engine Control Units), pacemakers, washing machines, and smart TVs, rely on

C for efficient processing.

- Since embedded systems have limited resources, C’s low memory footprint and direct hardware access make it the preferred choice.
- Many Internet of Things (IoT) devices run on microcontrollers programmed in C.

3. Game Development & Graphics Engines

Example: Unreal Engine, Doom, Quake

- Game engines like Unreal Engine use C for their core development due to its high performance.
- Classic games like Doom, Quake, and Counter-Strike were initially built using C, ensuring efficient memory management and real-time performance.
- Even modern game engines like Unity use C/C++ for performance-critical operations.

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.

Installing Dev-C++:

If you prefer an all-in-one package with an IDE and compiler, you can use Dev-C++, which includes GCC.

Step 1: Download Dev-C++

1. Go to the official Dev-C++ website: Dev-C++ Download.
2. Download the latest version of Dev-C++ (it includes MinGW automatically).

Step 2: Install Dev-C++

1. Run the downloaded .exe file.
2. Follow the installation wizard and choose default settings.
3. Open Dev-C++ after installation.

Step 3: Configure Dev-C++

1. Go to Tools → Compiler Options.
2. Ensure that the Compiler Set is set to TDM-GCC (which comes with Dev-C++).
3. Click OK to save changes.

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

||
||
||
||

DONE IN DEV C++

||
||
||
||

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.

1. Structure of a Simple C Program

A basic C program consists of:

1. Preprocessor Directives (Headers)
2. Main Function (main())
3. Comments
4. Variable Declarations and Data Types
5. Statements and Functions
6. Return Statement (if applicable)

1. **Preprocessor Directives (Headers)**

- Preprocessor directives begin with # and include necessary libraries.
- Example:

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

- #include <stdio.h> imports the Standard Input-Output library, enabling functions like printf() and scanf().

2. The main() Function (Entry Point)

- **Every C program must have a main() function where execution starts.**
- **Example:**

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

- int before main() indicates that the function returns an integer value.
- return 0; indicates successful execution (optional in some compilers).

3. Comments in C

- Single-line comments use // and explain specific parts of the code.
- Multi-line comments use /* */ for detailed explanations.
- Example:

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

4. Data Types and Variables

Common Data Types in C

--> TYPE--Size--Example

- int -- 4 (on most systems) -- int a = 5 ;
- float -- 4 -- float b = 3.14 ;
- double -- 8 -- double c = 9.8765 ;
- char -- 1 -- char d = **‘ A ’**

5. Printing Output Using printf()

- printf() is used to display output.
- Example:

printf("Age: %d\n", age); // %d is a format specifier for integers

Format Specifier	Data Type
%d	int
%f	float
%lf	double
%c	char
%s	String

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.

II
II
II
II

DONE IN DEV C++

II
II
II
II

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.

Arithmetic operators are used for mathematical calculations, including addition **(+)**, subtraction **(-)**, multiplication **(*)**, division **(/)**, and modulus **(%)** to find the remainder**.** For example, if **a = 10 and b = 5,** then **a + b** results in **15**, and **a % b** gives **0**.

Relational operators compare two values and return either true (1) or false (0). These include == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to). For example, if a = 10 and b = 5, then a > b results in 1 (true), while a == b gives 0 (false).

Logical operators are used for logical conditions, mainly && (logical AND), || (logical OR), and ! (logical NOT). If a = 1 and b = 0, then a && b evaluates to 0 (false), while a || b evaluates to 1 (true), and !a gives 0 (false).

Assignment operators assign values to variables and can be used in shorthand forms. The basic assignment operator is =, but combined forms like +=, -=, *=, /=, and %= allow operations and assignment in one step. For example, a += b is equivalent to a = a + b.

Increment and decrement operators increase or decrease a variable’s value by 1. The pre-increment (++a) and post-increment (a++) differ in their execution order. Similarly, pre-decrement (--a) and post-decrement (a--) behave differently depending on whether the value is updated before or after use in an expression.

Bitwise operators perform operations at the bit level and include **& (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), and >> (right shift).** For example, if a = 5 (binary 0101) and b = 3 (binary 0011), then a & b results in 1 (binary 0001), while a | b gives 7 (binary 0111).

conditional (ternary) operator (__?__ :__) is a shorthand for simple if-else statements. It takes the form **(condition ? true_value : false_value)** .
For example, min = (a < b) ? a : b; assigns the smaller value between a and b to min.

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.

II
II

DONE IN DEV C++

II
II

5. Control Flow Statements in C

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

1. if Statement

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

Syntax :

```
if (condition) {
    // Code executes if the condition is true
}
```

2. if-else Statement

The if-else statement allows two possible execution paths. If the condition is true, the if block executes; otherwise, the else block executes.

Syntax :

```
if (condition) {
    // Executes if condition is true
} else {
    // Executes if condition is false
}
```

3. Nested if-else Statement

A nested if-else statement is an if or else inside another if or else.

Syntax :

```
if (condition1) {
    if (condition2) {
        // Executes if both conditions are true
    } else {
        // Executes if condition1 is true but
condition2 is false
    }
} else {
    // Executes if condition1 is false
}
```

4. switch Statement

The switch statement is used when there are multiple possible values for a variable, and each value has a different execution path. It uses case labels to match values and an optional default for unmatched cases.

Syntax :

```
switch (expression) {
    case value1:
        // Code for value1
        break;
    case value2:
        // Code for value2
        break;
    default:
        // Code if no cases match
}
```

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.).

II

II

DONE IN DEV C++

II

II

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.

1. while Loop
The while loop checks the condition before executing the loop body. If the condition is true, it executes the block; otherwise, it skips it.
Syntax :

```
while (condition)
{
   // Code to execute
 }
```

When to Use?
--> When the number of iterations is not known in advance (e.g., reading input until the user enters a specific value).
--> Best for looping based on conditions rather than counters.

2. for Loop
The for loop is generally used when the number of iterations is known beforehand. It has an initialization, condition, and update in a single line.
Syntax :

```
for (initialization; condition; update)
{
   // Code to execute
}
```

When to Use?
--> When the number of iterations is fixed (e.g., looping through an array of known size).
--> Best for count-controlled loops where the loop variable is updated systematically.

3. do-while Loop
The do-while loop executes the loop body at least once, even if the condition is false because the condition is checked after execution.

Syntax :

```
do
{
   // Code to execute
}
while (condition);
```

When to Use?
--> When the loop body must execute at least once, regardless of the condition (e.g., menu-driven programs that display options before checking user input).

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

||
||

DONE IN DEV C++

||
||

7. Loop Control Statements

THEORY EXERCISE : Explain the use of break, continue, and go to statements in C. Provide examples of each.

1. break Statement

The break statement immediately exits a loop (for, while, do-while) or a switch statement.

EXAMPLE :

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // Exit loop when i equals 5
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output :

```
1 2 3 4
```

2. continue Statement

The continue statement skips the current iteration and moves to the next cycle of the loop.

EXAMPLE :

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            continue;  // Skip printing 5
        }
        printf("%d ", i);
    }
    return 0;
}
// it will skip number ' 5 '
```

OUTPUT :   1 2 3 4 6 7 8 9 10

3. goto Statement
The goto statement jumps to a labeled statement within the same function, altering program flow. It is not recommended as it makes debugging harder.
EXAMPLE :

```
#include <stdio.h>

int main() {
    int i = 1;

    loop_start: // Label
    printf("%d ", i);
    i++;

    if (i <= 5) {
        goto loop_start;  // Jump back to
loop_start
    }

    return 0;
}
```

OUTPUT :   1 2 3 4 5

LAB EXERCISE : Explain the use of break, continue, and go to statements in C. Provide examples of each.

||
||
||
||

DONE IN DEV C++

||
||
||
||

8. Functions in C

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

1. Function Declaration (Prototype)

A function must be declared before it is used. The declaration includes return type, function name, and parameters (if any).

Syntax :

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

Example :

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

2. Function Definition

The function definition provides the actual implementation of the function.

Syntax :

```
return_type function_name(parameter_list) {
    // Function body
    return value;  // If the function returns a
value
}
```

3. Function Call
To execute a function, it must be called in main() or another function.
Syntax :
```
function_name(arguments);
```

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

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

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

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

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

||
||
||
||

DONE IN DEV C++

||
||
||
||

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. It allows efficient data storage and manipulation.

1. One-Dimensional Array (1D Array)
A 1D array is a list of elements stored in a single row or single column.
Declaration & Initialization
--> data_type array_name[size];
--> int numbers[5] = {10, 20, 30, 40, 50}; // Integer array with 5 elements

EXAMPLE :

```
#include <stdio.h>

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

    // Printing array elements using a loop
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }

    return 0;
}
```

OUTPUT :

```
10  20  30  40  50
```

--> Best For : Storing a list of related values (e.g., marks, prices, temperatures).

2. Multi-Dimensional Arrays

A multi-dimensional array stores data in more than one dimension, such as 2D arrays (matrices) and 3D arrays.

--> data_type array_name[rows][columns];

--> int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };

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
```

--> Best For : Tables, grids, and matrices (e.g., representing a chessboard, pixel colors in images).

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.

||
||
||
||

DONE IN DEV C++

||
||
||
||

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?

Pointers in C are variables that store memory addresses of other variables. Instead of holding data directly, a pointer holds the location in memory where the data is stored.

1. Declaration of Pointers

To declare a pointer, use the **' * '** symbol :

data_type *pointer_name;

EXAMPLE :

```
int *ptr;   // Pointer to an integer
float *fptr; // Pointer to a float
```

2. Initialization of Pointers

A pointer is initialized using the address-of operator (&) which gives the memory address of a variable.

EXAMPLE :

```
int num = 10;
int *ptr = &num;  // 'ptr' now holds the address
of 'num'
```

3. Why Are Pointers Important in C?

1. **Dynamic Memory Management:**
   - Pointers enable functions like malloc() and free() for dynamic memory allocation.

2. **Efficient Array & String Handling:**
   - Arrays and strings in C are handled using pointers for faster access.

3. **Function Arguments (Pass by Reference):**
   - Pointers allow functions to modify variables directly, avoiding the need for return statements

4. **Data Structures Implementation:**
   - Pointers are crucial for implementing linked lists, trees, stacks, queues, etc.

5. **Memory-Level Control:**
   - C provides low-level memory access using pointers, making it powerful for system-level programming (e.g., OS development).

LAB EXERCISE : Write a C program to demonstrate pointer usage. Use a pointer to modify the value of a variable and print the result.

||
||
||
||

DONE IN DEV C++

||
||
||
||

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.

strlen() – String Length

- **Purpose:** Returns the length of a string (excluding the null character \0).
- **Syntax:** int len = strlen(str);
- **Example:**
  ```
  char str[] = "Hello";
   printf("Length: %lu", strlen(str));
  ```
- **Use Case:** Count characters in a string.

strcpy() – String Copy

- **Purpose:** Copies one string into another.
- **Syntax:** strcpy(dest, src);
- **Example:**
  ```
  char src[] = "World";
  char dest[10];
  strcpy(dest, src);
   printf("%s", dest);
  ```
- **Use Case:** Duplicate strings.

strcat() – String Concatenation

- **Purpose: Appends one string to the end of another.**
- **Syntax: strcat(dest, src);**
- **Example:**

```
char str1[20] = "Hello, ";
char str2[] = "World!";
strcat(str1, str2);
printf("%s", str1);
```

- **Use Case:** Combine two strings.

strcmp() – String Comparison

- **Purpose: Compares two strings.**
- **Returns:**
  - **0 → if strings are equal**
  - **<0 → if first string is less**
  - **>0 → if first string is greater**
- **Syntax: strcmp(str1, str2);**
- **Example:**

```
printf("%d", strcmp("abc", "abc"));
```

- **Use Case:** Sorting or checking equality.

strchr() – Find Character in String

- **Purpose: Returns a pointer to the first occurrence of a character in a string.**
- **Syntax: strchr(str, 'char');**
- **Example:**

```
char str[] = "hello";
char *ptr = strchr(str, 'e');
printf("%s", ptr);
```

- **Use Case:** Search for a character in a string.

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

II
II
II
II

DONE IN DEV C++

II
II
II
II

12. Structures in C

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

Structure (struct) in C is a user-defined data type that groups different data types under a single name.

1. Declaration:

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

2. Initialization:

```
struct Student s1 = {"Alice", 101, 95.5};
```

3. Accessing Members:

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

4. Using Pointers to Structures:

```
struct Student *ptr = &s1;
printf("Name: %s\n", ptr->name); // Use '->'
with pointers
```

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.

II
II
II
II

DONE IN DEV C++

II
II
II
II

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.

Opening a File:

Use fopen() with modes:

- "r" → Read
- "w" → Write (creates/overwrites)
- "a" → Append
- FILE *fp = fopen("data.txt", "w");

Writing to a File:

- Use fprintf() or fputs()
- fprintf(fp, "Hello, File!");

Reading from a File:

- Use fscanf(), fgets(), or fgetc()
- char buffer[50];
  fgets(buffer, sizeof(buffer), fp);

Closing a File:

- Use fclose() to free resources
- 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.

II
II
II
II

DONE IN DEV C++

II
II
II
II

THE END