PYTHON-BACKEND ASSIGNMENT

MODULE 2 – INTRODUCTION TO PROGRAMMING

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 History and Evolution of C Programming: Its Importance and Enduring Relevance**

    C programming is one of the most influential and foundational languages in the history of computer science. Its development, rooted in the late 1960s, represents a key turning point in programming, setting the stage for the evolution of modern software development. In this essay, we will explore the history and evolution of C programming, explain its importance, and discuss why it remains a vital tool in contemporary computing.

  - **Importance of C Programming:**
    - ✓ Portability and Efficiency.
    - ✓ System-Level Programming
    - ✓ Foundational Language for Modern Languages.
    - ✓ Widespread Adoption and Community Support.

  - **Why C Is Still Used Today:**
    - ✓ Embedded Systems and IoT:
    - ✓ Performance-Critical Applications:
    - ✓ Cross-Platform Development:
    - ✓ Modern Systems Software:
    - ✓ Simplicity and Transparency

- **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.

- **Embedded Systems:**
  - ✓ **Microcontrollers**: C is used to program microcontrollers, which are small computers embedded in products like smartphones, washing machines, or drones.
  - ✓ **Automotive Systems**: Modern vehicles contain numerous embedded systems for things like engine control units (ECUs), safety features (like airbags), and navigation systems.
- **Operating Systems:**
  - ✓ **Linux Kernel**: The Linux kernel, which forms the core of the Linux operating system, is largely written in C.
  - ✓ **Real-Time Operating Systems (RTOS)**: These are specialized operating systems used in critical applications such as medical devices, industrial control systems, and telecommunications, where real-time performance is crucial.
- **Game Development:**
  - ✓ **Game Engines**: Many game engines, including early versions of engines like Unity and Unreal Engine, use C for low-level system programming.
  - ✓ **Graphics and Performance Optimization**: C is used to write performance-critical sections of game engines, such as rendering engines and physics simulations, where fast execution and direct hardware access are vital.

2. Setting Up Environment.

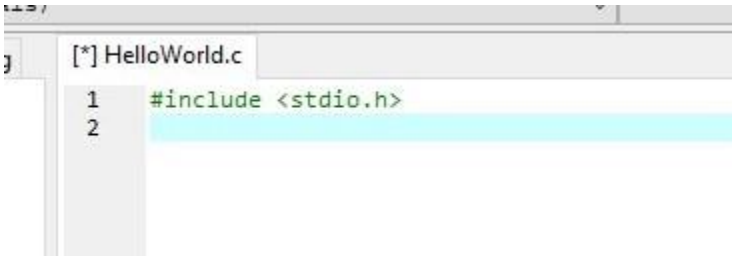
❖ **THEORY EXERCISE:** Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like DevC++, VS Code, or CodeBlocks.

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

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.

- **Headers:**
  - ✓ Headers in C are files that contain function declarations, constants, and macro definitions. These are included at the top of a C program using the #include preprocessor directive.
  - ✓ Example:



- **Main Function:**
  - ✓ Every C program must have a main function, which serves as the entry point of the program. The program execution begins from the main function. It can return an integer value, typically 0, to indicate successful execution.
  - ✓ Example:

- **Comments:**
  - ✓ Comments in C are used to annotate the code, explain its functionality, and improve readability.
  - ✓ Examples:

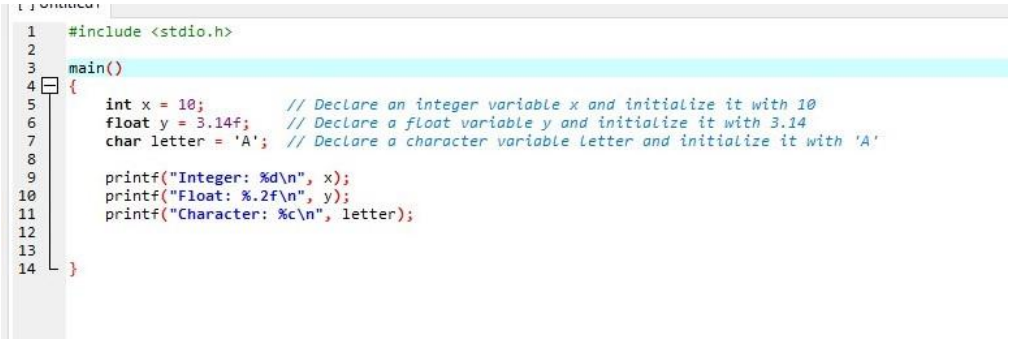
```c
#include <stdio.h>

int main() {
    // This is a single-line comment
    printf("Hello, World!\n"); // Prints a message to the console

    /* This is a multi-line comment
       that spans more than one line. */

}
```

- **Data Types:**
  - ✓ int: Integer type for whole numbers (positive or negative).
  - ✓ float: Floating-point type for numbers with decimals.
  - ✓ double: Double-precision floating-point type, offering more precision than float.
  - ✓ char: Character type for storing individual characters.
- **Variables:**
  - ✓ Variables are used to store data that can be modified during the execution of a program. To declare a variable, you must specify the data type followed by the variable name.
  - ✓ Example:



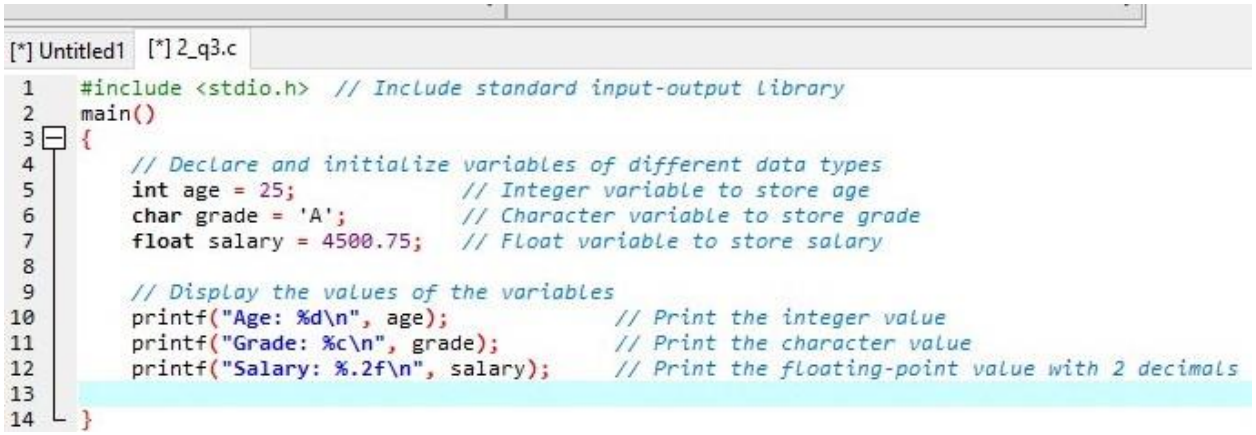
```c
#include <stdio.h>

main()
{
    int x = 10;         // Declare an integer variable x and initialize it with 10
    float y = 3.14f;    // Declare a float variable y and initialize it with 3.14
    char letter = 'A';  // Declare a character variable letter and initialize it with 'A'

    printf("Integer: %d\n", x);
    printf("Float: %.2f\n", y);
    printf("Character: %c\n", letter);


}
```

❖ **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.



[*] Untitled1 [*] 2_q3.c

```c
#include <stdio.h>  // Include standard input-output library
main()
{
    // Declare and initialize variables of different data types
    int age = 25;             // Integer variable to store age
    char grade = 'A';         // Character variable to store grade
    float salary = 4500.75;   // Float variable to store salary

    // Display the values of the variables
    printf("Age: %d\n", age);             // Print the integer value
    printf("Grade: %c\n", grade);         // Print the character value
    printf("Salary: %.2f\n", salary);     // Print the floating-point value with 2 decimals

}
```

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:**
  - ✓ **+ (Addition)**: Adds two operands.
    int sum = 5 + 3; // sum will be 8
  - ✓ **- (Subtraction)**: Subtracts the second operand from the first
    int difference = 5 - 3; // difference will be 2
  - ✓ *** (Multiplication)**: Multiplies two operands.
    int product = 5 * 3; // product will be 15
  - ✓ **/ (Division)**: Divides the first operand by the second
    int quotient = 5 / 3; // quotient will be 1 (integer division)
  - ✓ **% (Modulus)**: Returns the remainder when the first operand is divided by the second.
    int remainder = 5 % 3; // remainder will be 2

- **Relational Operators:**
  - ✓ **== (Equal to):** Returns true if two operands are equal.
  - ✓ **!= (Not equal to):** Returns true if two operands are not equal.
  - ✓ **> (Greater than):** Returns true if the left operand is greater than the right operand.
  - ✓ **< (Less than):** Returns true if the left operand is less than the right operand.
  - ✓ **>= (Greater than or equal to):** Returns true if the left operand is greater than or equal to the right operand.

- ✓ **<= (Less than or equal to):** Returns true if the left operand is less than or equal to the right operand.

- **Logical Operators:**
  - ✓ **&&(Logical AND):** Returns true if both operands are true.
    ```
    if (a > 0 && b > 0) {
     // true if both conditions are true
     }
    ```
  - ✓ **|| (Logical OR):** Returns true if at least one operand is true.
    ```
    if (a > 0 || b > 0) {
     // true if any condition is true
     }
    ```
  - ✓ **! (Logical NOT):** Reverses the logical state of its operand.
    ```
    if (!(a > 0)) {
    // true if a is not greater than 0
    }
    ```

- **Assignment Operators:**
  - ✓ **= (Simple assignment):** Assigns the value of the right operand to the left operand.
    ```
    int a = 5;  // a gets the value 5
    ```
  - ✓ **+= (Addition assignment):** Adds the right operand to the left operand and assigns the result to the left operand.
    ```
    a += 3;  // a = a + 3; (a = 8)
    ```
  - ✓ **-= (Subtraction assignment):** Subtracts the right operand from the left operand and assigns the result to the left operand.
    ```
    a -= 2;  // a = a - 2; (a = 6)
    ```
  - ✓ ***= (Multiplication assignment):** Multiplies the left operand by the right operand and assigns the result to the left operand.
    ```
    a *= 4;  // a = a * 4; (a = 24)
    ```
  - ✓ **/= (Division assignment):** Divides the left operand by the right operand and assigns the result to the left operand.
    ```
    a /= 2;  // a = a / 2; (a = 12)
    ```
  - ✓ **%= (Modulus assignment):** Takes the modulus of the left operand by the right operand and assigns the result to the left operand.
    ```
    a %= 5;  // a = a % 5; (a = 2)
    ```

- **Increment/Decrement Operators:**
  - ✓ **++ (Increment):** Increases the value of the operand by 1. It can be used as a *prefix* or *postfix*.
    ```
    a++;  // Post-increment: value of a is incremented after being used
    ++a;  // Pre-increment: value of a is incremented before being used
    ```

- ✓ **-- (Decrement):** Decreases the value of the operand by 1. It can also be used as a *prefix* or *postfix*.
  a--; // Post-decrement: value of a is decremented after being used
  --a; // Pre-decrement: value of a is decremented before being used

- **Conditional (Ternary) Operator:**
  - ✓ **? : (Ternary):** Evaluates the condition and returns one of two values. int result = (a > b) ? a : b; // returns a if a > b, otherwise returns b
  - ✓ The conditional operator works as:
    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.

```c
#include <stdio.h>

int main() {
    int a, b;
    // Accept two integers from the user
    printf("Enter the first integer: ");
    scanf("%d", &a);
    printf("Enter the second integer: ");
    scanf("%d", &b);
    // Arithmetic Operations
    printf("\nArithmetic 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);
    printf("Division (a / b): ");
    if (b != 0) {
        printf("%d\n", a / b);
    } else {
        printf("Division by zero is not allowed\n");
    }
    printf("Modulus (a %% b): ");
    if (b != 0) {
        printf("%d\n", a % b);
    } else {
        printf("Modulus by zero is not allowed\n");
    }
    // Relational Operations
    printf("\nRelational Operations:\n");
    printf("a == b: %d\n", a == b);  // 1 if true, 0 if false
    printf("a != b: %d\n", a != b); // 1 if true, 0 if false
    printf("a > b: %d\n", a > b);   // 1 if true, 0 if false
    printf("a < b: %d\n", a < b);   // 1 if true, 0 if false
    printf("a >= b: %d\n", a >= b); // 1 if true, 0 if false
    printf("a <= b: %d\n", a <= b); // 1 if true, 0 if false
    // Logical Operations
    printf("\nLogical Operations:\n");
    printf("a > 0 && b > 0: %d\n", (a > 0 && b > 0));  // Logical AND
    printf("a > 0 || b > 0: %d\n", (a > 0 || b > 0));  // Logical OR
    printf("!(a > 0): %d\n", !(a > 0));                // Logical NOT
}
```

5. Control Flow Statements in C

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

- **If Statement:**
  - ✓ **Syntax:**

    if (condition) {
    // Block of code to be executed if the condition is true
    }
  - ✓ **Example:**

```c
#include<stdio.h>
main()
{
    int mark;
    printf("\n\n\t enter the marks:");
    scanf("%d",&mark);
    if(mark >=40)
    {
        printf("pass");
    }
```

- **if-else Statement:**
  - ✓ **Syntax:**

    if (condition)
    {
    // Block of code to be executed if the condition is true
    }
    else
    {
    // Block of code to be executed if the condition is false
    }
  - ✓ **Example:**

```c
#include<stdio.h>
main()
{
    int mark;
    printf("\n\n\t enter the marks:");
    scanf("%d",&mark);
    if(mark >=40)
    {
        printf("pass");
    }
    else
    {

        printf("fail");
    }
}
```

- **if-else if-else:**
  - ✓ **Syntax:**

```
if (condition1) {
 // Block of code if condition1 is true
 }
 else if (condition2)
{
 // Block of code if condition2 is true
 }
 Else
 {
 // Block of code if none of the above conditions are true
 }
```

  - ✓ **Example:**

```c
#include<stdio.h>
main()
{
    int a,b;
    char choice;
    printf("enter the number1 :");
    scanf("%d",&a);
    printf("enter the number2 :");
    scanf("%d",&b);
    printf("\n------------------------");
    printf("\n press + for addition");
    printf("\n press - for subtraction");
    printf("\n press * for product");
    printf("\n press / for division");
    printf("\n------------------------");
    printf("\n select option");
    scanf(" %c",&choice);

    if(choice=='+')
        printf("addition:%d",a+b);
    else if(choice=='-')
        printf("subtraction:%d",a-b);
    else if(choice=='*')
        printf("product:%d",a*b);
    else if(choice=='/')
        printf("division:%d",a/b);
    else
        printf("\n valid option choice");


}
```

- **Nested if-else Statements:**
  - ✓ **Syntax:**

```
if (condition1) {
  if (condition2) {
    // Block of code if condition1 and condition2 are true
  } else {
    // Block of code if condition1 is true but condition2 is false
  }
} else {
  // Block of code if condition1 is false
}
```

  - ✓ **Example:**

```c
#include<stdio.h>
main()
{
    int a,b,c;
    printf("enter any three nmber :");
    scanf("%d %d %d",&a,&b,&c);
    if(a>b)
    {
        if(a>c)
        {
            printf("\n\n\t %d is max",a);
        }
    }
    if(b>a)
    {
    if(b>c)
        {
            printf("\n\n\t %d is max",b);
        }

    }
    if(c>a)
    {
    if(c>a)
        {
            printf("\n\n\t %d is max",c);
        }

    }
}
```

- **switch Statement:**
  - ✓ **Syntax:**

```
switch (expression) {
  case value1:
    // Code to execute if expression == value1
    break;
  case value2:
    // Code to execute if expression == value2
    break;
  // Add more cases as needed
  default:
    // Code to execute if no case matches
}
```

  - ✓ **Example:**

```c
#include<stdio.h>
main()
{
    int a,b;
    char choice;
    printf("enter the number1 :");
    scanf("%d",&a);
    printf("enter the number2 :");
    scanf("%d",&b);
    printf("\n-----------------------");
    printf("\n press + for addition");
    printf("\n press - for subtraction");
    printf("\n press * for product");
    printf("\n press / for division");
    printf("\n-----------------------");
    printf("\n select option");
    scanf(" %c",&choice);

    switch(choice)
    {
        case '+' : printf("\n\n\t addition :%d",a+b);
                   break;
        case '-' : printf("\n\n\t subtraction :%d",a-b);
                   break;
        case '*' : printf("\n\n\t product :%d",a*b);
                   break;
        case '/' : printf("\n\n\t division :%d",a/b);
                   break;
        default : printf("\n\n\t invalide choice");
                     break;


    }
}
```

- **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.).
    - Write a C program to check if a number is even or odd using an if-else statement.

```c
//odd or even
#include<stdio.h>
main()
{
    int a;
    printf("enter number  :");
    scanf("%d",&a);
    printf("\n-----------------------\n\n");

    if(a%2==0)
    {
        printf("even");
    }
    else
    {
        printf("odd");
    }

}
```

- 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.).

```c
#include <stdio.h>

int main() {
    int month;
    printf("Enter a number between 1 and 12 to display the month: ");
    scanf("%d", &month);
    switch (month) {
        case 1:printf("January\n");
            break;
        case 2:printf("February\n");
            break;
        case 3:printf("March\n");
            break;
        case 4:printf("April\n");
            break;
        case 5:printf("May\n");
            break;
        case 6:printf("June\n");
            break;
        case 7:printf("July\n");
            break;
        case 8:printf("August\n");
            break;
        case 9:printf("September\n");
            break;
        case 10:printf("October\n");
            break;
        case 11:printf("November\n");
            break;
        case 12:printf("December\n");
            break;
        default:printf("Invalid input! Please enter a number between 1 and 12.\n");
    }
}
```

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.

- **Comparison and Contrast of While Loops**, **For Loops,** and **Do-While Loops:**
  - **While Loop :**
    - ✓ **Syntax**:
      while condition:
        # Code to execute
    - ✓ **Description**: A while loop executes a block of code repeatedly as long as the specified condition evaluates to True.

- ✓ **When to Use**: **Scenarios**: Ideal for situations like input validation or when iterating over dynamic data (e.g., reading from a file until it's empty).

- **For Loop:**
  - ✓ **Syntax**:
    for variable in iterable:
      # Code to execute
  - ✓ **Description:** A for loop iterates over a sequence (like a list, string, range, or any iterable) and executes the code block once for each element.
  - ✓ **When to Use : Fixed number of iterations**: Use when you know the number of iterations ahead of time, such as iterating over a list or a range of numbers.
- **Do-While Loop:**
  - ✓ **Syntax:**
    do {
      // Code to execute
    } while (condition);
  - ✓ **Description :** A **do-while loop** guarantees that the code block will execute at least once before the condition is checked.
  - ✓ **When to Use : Guaranteed first execution**: Use when you want the loop body to run at least once regardless of the condition, and you need to check the condition after the first execution.

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

```c
#include <stdio.h>

int main() {
    // Using While Loop
    printf("Using While Loop:\n");
    int i = 1;
    while (i <= 10) {
        printf("%d ", i);
        i++;
    }
    printf("\n\n");

    // Using For Loop
    printf("Using For Loop:\n");
    for (i = 1; i<=10; i++)
     {
        printf("%d ", i);
    }
    printf("\n\n");

    // Using Do-While Loop
    printf("Using Do-While Loop:\n");
    int j = 1;
    do {
        printf("%d ", j);
        j++;
    } while (j <= 10);
    printf("\n");
}
```

7. Loop Control Statements.

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

- **break Statement:** The break statement is used to exit from a loop or switch-case statement prematurely. When a break is encountered, the program control is transferred immediately to the statement following the loop or switch block.

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // Exit the loop when i equals 5
        }
        printf("%d ", i);  // Output: 1 2 3 4
    }

}
```

- **continue Statement** : The continue statement is used to skip the current iteration of a loop and proceed with the next iteration. When continue is executed, the remaining code in the loop for that iteration is skipped, and control is transferred to the loop's condition-checking statement.

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;  // Skip the iteration when i equals 3
        }
        printf("%d ", i);  // Output: 1 2 4 5
    }

}
```

- **goto Statement :** The goto statement is used to transfer control to another part of the program. It is often considered a dangerous or poor practice because it can make code

harder to follow and maintain. However, it can be useful for certain situations, such as error handling or breaking out of deeply nested loops.

```c
#include <stdio.h>
main()
{
    int i = 0;
    start:
        if (i < 5)
        {
            printf("%d ", i);  // Output: 0 1 2 3 4
            i++;
            goto start;  // Jump back to the label "start"
        }
}
```

❖ **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 using break statement :**

```c
#include <stdio.h>
main()
{
    for (int i = 1; i <= 10; i++)
    {
        if (i == 5)
        {
            break;  // Exit the loop when i equals 5
        }
        printf("%d ", i);  // Output will be 1 2 3 4
    }
}
```

- **Modify the program to skip printing the number 3 using continue statement :**

```c
#include <stdio.h>
main()
{
    for (int i = 1; i <= 10; i++)
    {
        if (i == 3)
        {
            continue;  // Skip printing the number 3
        }
        printf("%d ", i);  // Output will be 1 2 4 5 6 7 8 9 10
    }
}
```

8. Functions in C

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

- **Functions in C** : A function in C is a block of code that performs a specific task. Functions help to break a program into smaller, manageable pieces, promote code reusability, and improve readability and maintainability.

- **Function Declaration** (also called Function Prototype) : The function declaration specifies the function's name, return type, and parameters (if any), but it does not include the function body.
  - ✓ Syntax:
    return_type function_name(parameter1_type parameter1, parameter2_type parameter2, ...);

- **Function Definition :** The function definition provides the actual body of the function, where the task or logic is implemented.
  - ✓ Syntax:
    return_type function_name(parameter1_type parameter1, parameter2_type parameter2, ...)
    {
    // Code to be executed

}

- **Function Call :** A function is called by using its name and passing the required arguments (if any) that match the parameter types defined in the function.
  - ✓ Syntax:
    function_name(argument1, argument2, ...);

- **Example :**

```c
#include <stdio.h>
int add(int, int);
int main() {
    int result;
    result = add(5, 3);
    printf("The sum is: %d\n", result);
    return 0;
}
int add(int num1, int num2) {
    return num1 + num2;
}
```

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

```c
#include <stdio.h>
long long factorial(int n);
int main() {
    int number;
    long long result;
    printf("Enter a number to calculate its factorial: ");
    scanf("%d", &number);
    result = factorial(number);
    printf("The factorial of %d is: %lld\n", number, result);
    return 0;
}
long long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

9. Arrays in C

❖ **THEORY EXERCISE** : Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

- **Arrays in C :**
  - ✓ An **array** in C is a collection of variables of the same type that are stored in contiguous memory locations. Arrays allow you to store multiple values under a single variable name, and you can access each value using an index.
  - ✓ Types of Arrays :
    - ➢ **One-Dimensional Array**: A simple array that holds a single row of values.
    - ➢ **Multi-Dimensional Array**: Arrays that can hold more than one row/column of values (like a matrix or a table).

- **Differentiate between one-dimensional and multi-dimensional arrays :**

Feature	**One-Dimensional Array**	**Multi-Dimensional Array**
Shape	A single row of values.	Multiple rows and columns (2D), or even higher dimensions.
Accessing elements	Accessed using a single index.	Accessed using multiple indices (row, column, etc.).
Declaration	data_type array_name[size];	data_type array_name[rows][columns];
Example	int arr[5] = {1, 2, 3, 4, 5};	int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

❖ **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.

```c
#include <stdio.h>
main() {
    int i;
    int arr[5];
    printf("Enter 5 integers for the one-dimensional array:\n");
    for (i = 0; i < 5; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
    printf("\nThe one-dimensional array is:\n");
    for (i = 0; i < 5; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }
    int j;
    int matrix[3][3];
    int sum = 0;
    printf("\nEnter elements for the 3x3 matrix:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("Element [%d][%d]: ", i + 1, j + 1);
            scanf("%d", &matrix[i][j]);
        }
    }
    printf("\nThe 3x3 matrix is:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            sum += matrix[i][j];
        }
    }
    printf("\nThe sum of all elements in the 3x3 matrix is: %d\n", sum);
    return 0;
}
```

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 :** A **pointer** in C is a variable that stores the memory address of another variable. Instead of holding a data value directly (like an integer or a character), a pointer holds the address of the location in memory where the data is stored. Pointers are an essential feature of C because they provide a powerful mechanism for directly accessing and manipulating memory.

- **Why Pointers Are Important in C :**
  - ✓ **Memory Efficiency**: Pointers allow efficient memory management by directly manipulating memory locations instead of working with large amounts of data.

- ✓ **Dynamic Memory Allocation**: With pointers, you can dynamically allocate memory (using functions like malloc() and free()), which is useful for programs that need to work with variable-sized data structures.
- ✓ **Function Arguments**: Pointers are used to pass arguments to functions by reference, allowing functions to modify variables outside of their scope (instead of passing copies of variables).
- ✓ **Arrays and Strings**: In C, arrays are closely tied to pointers, and pointers can be used to work with arrays and strings more flexibly.

- **Pointer Declaration and Initialization :**

```c
#include <stdio.h>
main() {
    int x = 10;
    int *ptr = &x;
    printf("Value of x: %d\n", x);
    printf("Value of x via pointer ptr: %d\n", *ptr);

    printf("Address of x: %p\n", &x);
    printf("Address stored in ptr: %p\n", ptr);
    return 0;
}
```

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

```c
#include <stdio.h>
main() {
    int num = 10;
    int *ptr = &num;
    printf("Before modification:\n");
    printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", (void*)&num);
    printf("Value stored in ptr (address of num): %p\n", (void*)ptr);
     *ptr = 20;
    printf("\nAfter modification using pointer:\n");
    printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", (void*)&num);
    printf("Value stored in ptr (address of num): %p\n", (void*)ptr);
    return 0;
}
```

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()**: Get the length of a string
  - ✓ The strlen() function returns the number of characters in a string, excluding the null character ('\0').
  - ✓ **Syntax :**
    size_t strlen(const char *str);
  - ✓ **Example :**

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

int main() {
    char str[] = "Hello, World!";
    int length = strlen(str);
    printf("Length of the string: %d\n", length);
    return 0;
}
```

- **strcpy()**: Copy a string
  - ✓ The strcpy() function copies the content of one string into another.
  - ✓ **Syantax :**
    char *strcpy(char *dest, const char *src);
  - ✓ **Example :**

```c
#include <stdio.h>
main() {
    char source[] = "Hello, World!";
    char destination[50];
    strcpy(destination, source);
    printf("Source: %s\n", source);
    printf("Destination: %s\n", destination);
    return 0;
}
```

- **strcat()**: Concatenate two strings
  - ✓ The strcat() function appends one string to the end of another string.
  - ✓ **Syntax :**
    char *strcat(char *dest, const char *src);

✓ **Example :**

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

int main() {
    char str1[50] = "Hello, ";
    char str2[] = "World!";

    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

- **strcmp()**: Compare two strings
  - ✓ The strcmp() function compares two strings lexicographically (character by character).
  - ✓ **Syntax :**

    int strcmp(const char *str1, const char *str2);
  - ✓ **Example :**

```c
#include <stdio.h>
main() {
    char str1[] = "Apple";
    char str2[] = "Banana";

    int result = strcmp(str1, str2);
    if (result < 0) {
        printf("\"%s\" is less than \"%s\"\n", str1, str2);
    } else if (result > 0) {
        printf("\"%s\" is greater than \"%s\"\n", str1, str2);
    } else {
        printf("Both strings are equal\n");
    }
    return 0;
}
```

- **strchr()**: Find the first occurrence of a character in a string
  - ✓ The strchr() function searches for the first occurrence of a specified character in a string.
  - ✓ **Syntax :**

    char *strchr(const char *str, int c);
  - ✓ **Example :**

```c
#include <stdio.h>
main() {
    char str[] = "Hello, World!";
    char *result;
    result = strchr(str, 'W');
    if (result != NULL) {
        printf("Character found: %c\n", *result);
    } else {
        printf("Character not found.\n");
    }
    return 0;
}
```

❖ **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().

```c
#include <stdio.h>
main()
{
    char str1[100], str2[100];
    printf("Enter the first string: ");
    fgets(str1, sizeof(str1), stdin);
    str1[strcspn(str1, "\n")] = '\0';
     printf("Enter the second string: ");
    fgets(str2, sizeof(str2), stdin);
    str2[strcspn(str2, "\n")] = '\0';
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    printf("Length of concatenated string: %zu\n", strlen(str1));
}
```

12. Structures in C :

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

- **Concept of Structures in C :** A **structure** in C is a user-defined data type that allows grouping of different types of variables (called members or fields) under a single name. Each member of a structure can have a different data type (e.g., integers, floats, arrays, or even other structures).

- **Declaring a Structure :** To define a structure, you use the struct keyword followed by the structure name and its members inside curly braces.

  Here is the syntax to declare a structure:

```
struct structure_name {
        data_type member1;
        data_type member2;
        // More members
};
```

- **Initializing a Structure :** There are two ways to initialize a structure :

  - ✓ **At the time of declaration (Static Initialization)**: You can initialize the structure members when you declare a structure variable:

    ```
    struct Student student1 = {"John Doe", 20, 85.5};
    ```

  - ✓ **After Declaration (Dynamic Initialization)**: You can declare the structure first and then assign values to its members individually:

    ```
    struct Student student1;
    strcpy(student1.name, "John Doe");  // Using strcpy for string assignment
    student1.age = 20;
    student1.marks = 85.5;
    ```

- **Accessing Structure Members :** Structure members can be accessed using the **dot operator** (.) for individual structure variables. If the structure variable is a pointer, you use the **arrow operator** (->) to access the members.

❖ **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.

```c
#include <stdio.h>
struct Student {
    char name[50];
    int rollNumber;
    float marks;
};

main() { int i;
    struct Student students[3];
    for (i = 0; i < 3; i++) {
        printf("Enter details for student %d:\n", i + 1);
        printf("Enter name: ");
        fgets(students[i].name, sizeof(students[i].name), stdin);
        students[i].name[strcspn(students[i].name, "\n")] = '\0';
        printf("Enter roll number: ");
        scanf("%d", &students[i].rollNumber);
        printf("Enter marks: ");
        scanf("%f", &students[i].marks);
        getchar();
    }
    printf("\nStudent Details:\n");
    for (i = 0; i < 3; i++) {
        printf("\nStudent %d:\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].rollNumber);
        printf("Marks: %.2f\n", students[i].marks);
    }
}
```

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.
    - **Importance of File Handling in C :**
        - ✓ Persistent Data Storage:
        - ✓ Data Sharing:
        - ✓ Efficient Data Management:
        - ✓ Flexible Data Operations:
    - **Basic File Operations in C :**
        - ✓ Opening a File : To perform any operation on a file, you first need to open the file using the fopen() function. This function requires two arguments 1=The name of the file. 2=The mode in which you want to open the file.
            - ➢ Syntax :
              FILE *fopen(const char *filename, const char *mode);
        - ✓ Reading from a File : Once the file is opened in read mode, you can read from it using functions like fgetc(), fgets(), or fread().
        - ✓ Writing to a File : You can write data to a file using fputc(), fputs(), or fwrite().

- ✓ **Closing a File :** After performing the required operations (read or write), it's essential to close the file using fclose() to release resources and ensure data integrity

- **Example :**

```c
#include <stdio.h>
main() {
    FILE *file;
    char buffer[255];
    // Open a file in write mode
    file = fopen("example.txt", "w");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }
    // Write data to the file
    fprintf(file, "This is a test file.\n");
    fputs("Hello, world!\n", file);
    fclose(file);  // Close the file after writing
    // Open the file in read mode
    file = fopen("example.txt", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }
    // Read data from the file and display it
    printf("File content:\n");
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("%s", buffer);
    }
    fclose(file);  // Close the file after reading

    return 0;
}
```

❖ **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.

```c
#include <stdio.h>

int main() {
    FILE *file;
    char *text = "Hello, this is a sample string written to the file.";
    file = fopen("sample.txt", "w");
    if (file == NULL) {
        printf("Error opening file for writing.\n");
        return 1;
    }
    fprintf(file, "%s", text);
    printf("String written to the file successfully.\n");
    fclose(file);
    file = fopen("sample.txt", "r");
    if (file == NULL) {
        printf("Error opening file for reading.\n");
        return 1;
        char ch;
    printf("\nReading from the file:\n");
    while ((ch = fgetc(file)) != EOF) {
    putchar(ch);
    fclose(file);
    return 0;
}
```