Module -3

Introduction to OOPS Programming

1. Introduction to C++

THEORY EXERCISE:

- **What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?**

Aspect	**Procedural Programming (PP)**	**Object-Oriented Programming (OOP)**
Focus	Procedures/functions	Objects (data and behavior)
Data and Functions	Separate	Combined in objects
Approach	Top-down (step-by-step)	Bottom-up (objects interacting)
Modularity	Limited, based on functions	High, based on classes and objects
Inheritance	No inheritance	Supports inheritance for code reuse

- **List and explain the main advantages of OOP over POP.**

Advantage	**OOP**	**POP**
Modularity	Code is organized into classes/objects	Code is organized into functions/procedures
Code Reusability	Inheritance and polymorphism for reuse	Limited reuse through function calls
Data Encapsulation	Data is hidden within objects	Data is often global, accessible by any function
Maintainability	Easier to maintain due to modularity	Harder to maintain with large codebases

Scalability	Easy to scale with inheritance and extensions	Scaling requires major restructuring

- **Explain the steps involved in setting up a C++ development environment.**
  - Install a C++ Compiler
  - Install a Code Editor or Integrated Development Environment (IDE)
  - Set Up the Environment (Configure PATH)
  - Test the Compiler
  - Optional: Install Debugging Tools
  - Install Libraries (Optional)
  - Start Writing C++ Code

- **What are the main input/output operations in C++? Provide examples.**

 Main Input/Output Operations in C++:

1.Standard Input (cin) :

✓ **`cin`** is the standard input stream used to take input from the user (usually from the keyboard). ✓ It is a part of the `iostream` library. **2.** Standard Output (cout) :

✓ **`cout`** is the standard output stream used to display output to the user (usually on the screen).

2. Variables, Data Types, and Operators :

THEORY EXERCISE:

- **What are the different data types available in C++? Explain with examples.**

➢ **Primitive data types**

- **int**: Stores whole numbers without decimal points
- **char**: Stores individual characters, letters, or ASCII values

- **float**: Stores decimal numbers with fewer digits
- **double**: Stores decimal numbers with more digits
- **bool**: Stores Boolean values
- **void**: An incomplete type used for functions that do not return a value

➢ **Derived data types**

- **Array**: Derived from primitive data types
- **Function**: Derived from primitive data types
- **Pointer**: Derived from primitive data types
- **Reference**: Derived from primitive data types

➢ **User-defined data types**

- **Class**: Defined by the user
- **Structure**: Defined by the user
- **Union**: Defined by the user
- **Enumeration**: Also known as enum, this data type consists of a set of named constants

▪ Explain the difference between implicit and explicit type conversion in C++.

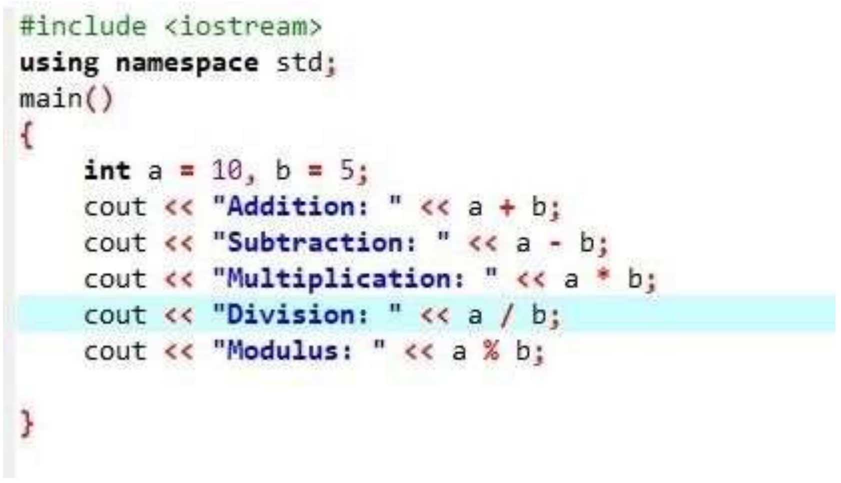
Feature	**Implicit Type Conversion**	**Explicit Type Conversion**
Initiated By	Automatically done by the compiler.	Manually performed by the programmer.
Type of Conversion	From lower precision to higher precision (e.g., int to float).	Can be from higher precision to lower precision (e.g., double to int).
Use Case	When there's no risk of data loss or errors.	When data loss may occur or the programmer needs to control the conversion.
Syntax	No special syntax is needed.	Requires a cast operator ((type), static_cast<type>, etc.).
Safety	Generally safe if done automatically.	Can be unsafe if not carefully done (e.g., truncating values).

▪ What are the different types of operators in C++? Provide examples of each.

➢ **Arithmetic Operators** : These operators are used to perform basic mathematical operations.

- **+ (Addition)**: Adds two operands.

- **(Subtraction)**: Subtracts the second operand from the first.
- **(Multiplication)**: Multiplies two operands.
- **/ (Division)**: Divides the numerator by the denominator.
- **% (Modulus)**: Returns the remainder when one operand is divided by the other.

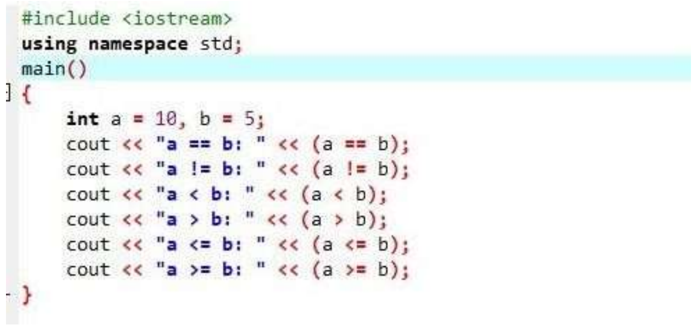


```
#include <iostream>
using namespace std;
main()
{
    int a = 10, b = 5;
    cout << "Addition: " << a + b;
    cout << "Subtraction: " << a - b;
    cout << "Multiplication: " << a * b;
    cout << "Division: " << a / b;
    cout << "Modulus: " << a % b;

}
```

➢ **Relational (Comparison) Operators :** These operators are used to compare two values.

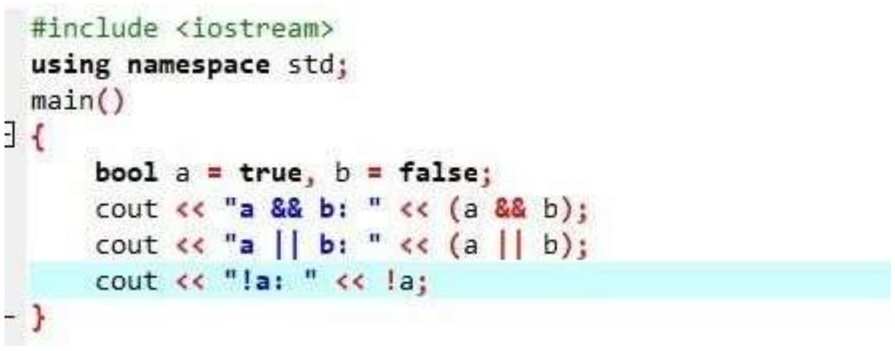
- **== (Equal to)**: Checks if two values are equal.
- **!= (Not equal to)**: Checks if two values are not equal.
- **< (Less than)**: Checks if the left value is less than the right value.
- **> (Greater than)**: Checks if the left value is greater than the right value.
- **<= (Less than or equal to)**: Checks if the left value is less than or equal to the right value.
- **>= (Greater than or equal to)**: Checks if the left value is greater than or equal to the right value.



```
#include <iostream>
using namespace std;
main()
{
    int a = 10, b = 5;
    cout << "a == b: " << (a == b);
    cout << "a != b: " << (a != b);
    cout << "a < b: " << (a < b);
    cout << "a > b: " << (a > b);
    cout << "a <= b: " << (a <= b);
    cout << "a >= b: " << (a >= b);
}
```

➢ **Logical Operators :** These operators are used to perform logical operations, commonly with Boolean values.

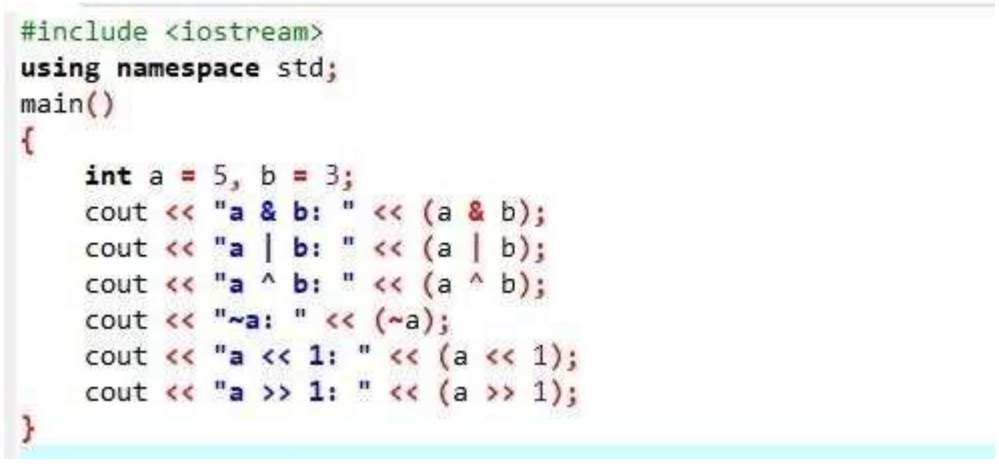
- **&& (Logical AND)**: Returns true if both operands are true.
- **|| (Logical OR)**: Returns true if at least one operand is true.
- **! (Logical NOT)**: Reverses the logical state of its operand. If the condition is true, it becomes false, and vice versa.



```
#include <iostream>
using namespace std;
main()
{
    bool a = true, b = false;
    cout << "a && b: " << (a && b);
    cout << "a || b: " << (a || b);
    cout << "!a: " << !a;
}
```

➢ **Bitwise Operators :** These operators perform bit-level operations on integers.

- **& (Bitwise AND)**: Performs bitwise AND operation.
- **| (Bitwise OR)**: Performs bitwise OR operation.
- **^ (Bitwise XOR)**: Performs bitwise XOR operation.
- **~ (Bitwise NOT)**: Inverts all the bits of its operand.
- **<< (Left shift)**: Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
- **>> (Right shift)**: Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

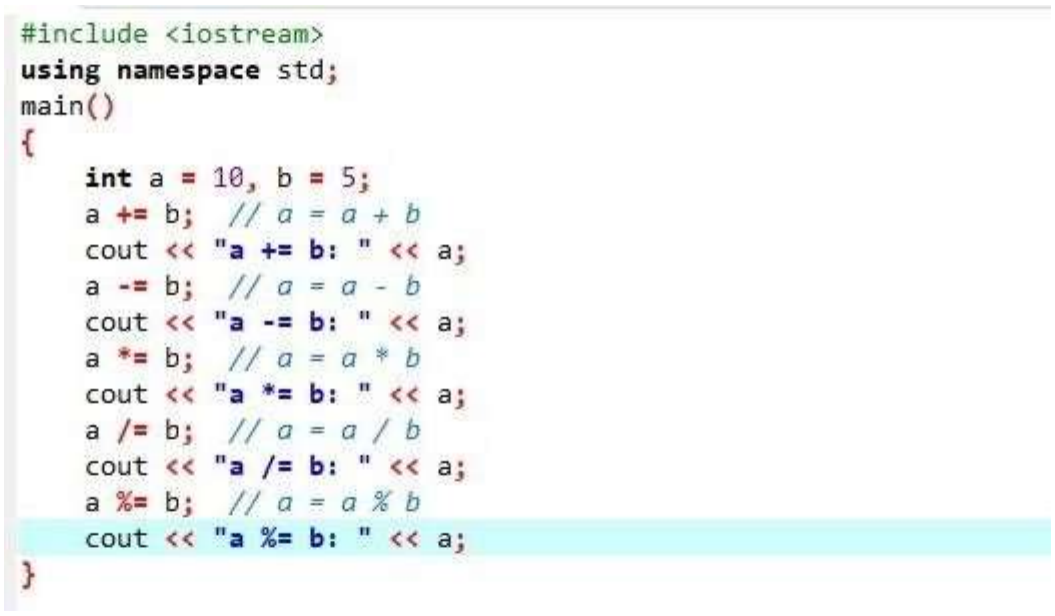


```
#include <iostream>
using namespace std;
main()
{
    int a = 5, b = 3;
    cout << "a & b: " << (a & b);
    cout << "a | b: " << (a | b);
    cout << "a ^ b: " << (a ^ b);
    cout << "~a: " << (~a);
    cout << "a << 1: " << (a << 1);
    cout << "a >> 1: " << (a >> 1);
}
```

➢ **Assignment Operators :** These operators are used to assign values to variables.

- **=**: Simple assignment.
- **+=**: Adds the right operand to the left operand and assigns the result to the left operand.
- **-=**: Subtracts the right operand from the left operand and assigns the result to the left operand.

- ***=**: Multiplies the left operand by the right operand and assigns the result to the left operand.
- **/=**: Divides the left operand by the right operand and assigns the result to the left operand.
- **%=**: Takes the modulus of the left operand by the right operand and assigns the result to the left operand.



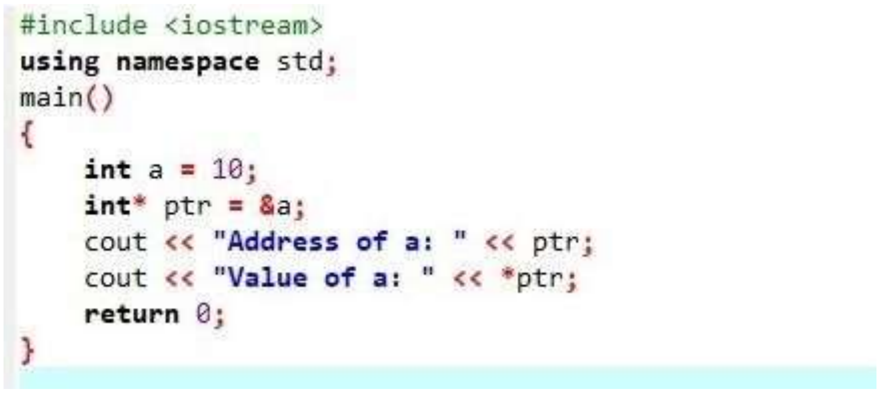
```
#include <iostream>
using namespace std;
main()
{
    int a = 10, b = 5;
    a += b;  // a = a + b
    cout << "a += b: " << a;
    a -= b;  // a = a - b
    cout << "a -= b: " << a;
    a *= b;  // a = a * b
    cout << "a *= b: " << a;
    a /= b;  // a = a / b
    cout << "a /= b: " << a;
    a %= b;  // a = a % b
    cout << "a %= b: " << a;
}
```

➢ **Unary Operators :** These operators operate on a single operand.

- **++ (Increment)**: Increases the value of the operand by 1.
- **-- (Decrement)**: Decreases the value of the operand by 1.
- **+ (Unary plus)**: Indicates a positive value.
- **- (Unary minus)**: Negates the value.
- **! (Logical NOT)**: Reverses the Boolean value.

➢ **Pointer Operators :** These operators are used to work with pointers.

- **& (Address-of operator)**: Returns the memory address of a variable.
- *** (Dereference operator)**: Accesses the value at the memory address stored in a pointer.



```
#include <iostream>
using namespace std;
main()
{
    int a = 10;
    int* ptr = &a;
    cout << "Address of a: " << ptr;
    cout << "Value of a: " << *ptr;
    return 0;
}
```

- **Explain the purpose and use of constants and literals in C++.**

➢ **Constants in C++ :** A **constant** is a value that **cannot be changed** once it has been initialized. Constants are used to define values that are intended to remain the same throughout the execution of the program.

➢ **Purpose of constants :**

- **Improved Readability**: Constants make the code more readable and easier to understand, as they give meaningful names to fixed values.
- **Avoiding Errors**: Constants prevent accidental modification of values that should not change, reducing the risk of bugs.
- **Easier Maintenance**: If a constant value is needed in multiple places in the code, it can be modified in just one place, making the program easier to maintain.

➢ **Purpose of Literals:**

- **Efficiency**: Literals provide a simple way to represent constant values directly in the code, making them efficient to use.
- **Simplify Code**: Using literals instead of variables for simple constants helps simplify code and avoid unnecessary variable declarations.

3. Control Flow Statements :

THEORY EXERCISE:

▪ What are conditional statements in C++? Explain the if-else and switch statements.

In C++, **conditional statements** allow you to execute certain blocks of code based on whether a specified condition is true or false. These control structures are used to make decisions in a program. C++ provides several types of conditional statements, with the most common being the if-else, and switch statements.

➢ if-else Statement

The if-else statement allows for two blocks of code: one that gets executed if the condition is true, and another that gets executed if the condition is false. This is useful when you need to choose between two options based on a condition.

- Syntax :

```
if (condition) {

   // Code block to be executed if the condition is true

} else {

   // Code block to be executed if the condition is false

}
```

➢ switch Statement

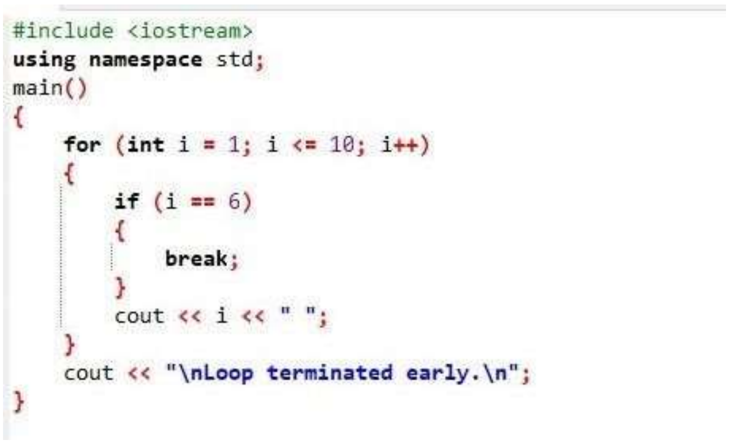
The switch statement is another type of conditional statement that is used when you have multiple possible values for a single variable and want to choose between them. It is often used as an alternative to long if-else if chains when there are many possible conditions.

▪ What is the difference between for, while, and do-while loops in C++?

Feature	**for Loop**	**while Loop**	**do-while Loop**
Condition check	Before each iteration	Before each iteration	After each iteration

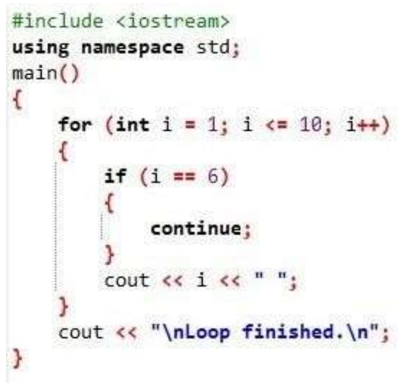
Initialization	Can be done at the start of the loop	Done outside the loop	Done outside the loop
Increment/Decrement	Done in the loop	Done inside the loop	Done inside the loop
	itself		
Use case	Iterating over a known range or count (e.g., array)	For conditions where you don't know the number of iterations	When at least one iteration is required, regardless of condition

- **How are break and continue statements used in loop? Provide example.**
  - **break Statement :** The **break** statement is used to **exit** a loop prematurely. When the break statement is encountered, the loop is terminated immediately, and the program continues executing the code that follows the loop.



```
#include <iostream>
using namespace std;
main()
{
    for (int i = 1; i <= 10; i++)
    {
        if (i == 6)
        {
            break;
        }
        cout << i << " ";
    }
    cout << "\nLoop terminated early.\n";
}
```

  - **continue Statement :**The continue statement is used **to** skip the current iteration of the loop and move to the next iteration. When the continue statement is encountered, the remaining code in the current iteration is skipped, and the loop proceeds with the next iteration.



```
#include <iostream>
using namespace std;
main()
{
    for (int i = 1; i <= 10; i++)
    {
        if (i == 6)
        {
            continue;
        }
        cout << i << " ";
    }
    cout << "\nLoop finished.\n";
}
```

- **Explain nested control structures with an example.**

  - Types of Nested Control Structures :

    ✓ **Nested `if` Statements**: An `if` statement inside another `if` or `else` block.

    ✓ **Nested Loops**: A loop inside another loop (either `for`, `while`, or `do-while`).

    ✓ **Mixed Nesting**: A combination of conditionals inside loops, or loops inside conditionals.

4. Functions and Scope :

THEORY EXERCISE:

- **What is a function in C++? Explain the concept of function declaration, definition, and calling.**

A function in C++ is a block of code that performs a specific task and can be executed when called by name. Functions allow us to break down complex problems into smaller, manageable parts and reuse code.

➢ **Function Declaration**: A function declaration (or prototype) tells the compiler about the function's name, return type, and parameters (if any), but does not include the body (actual code implementation).

- **Syntax :**
  return_type function_name(parameter_list);

➢ **Function Definition**: The function definition provides the actual body of the function. It specifies what the function does with the parameters and how it returns a value (if applicable).

- **Syntax :** return_type function_name(parameter_list)
  {

```
        // function body
    }
```

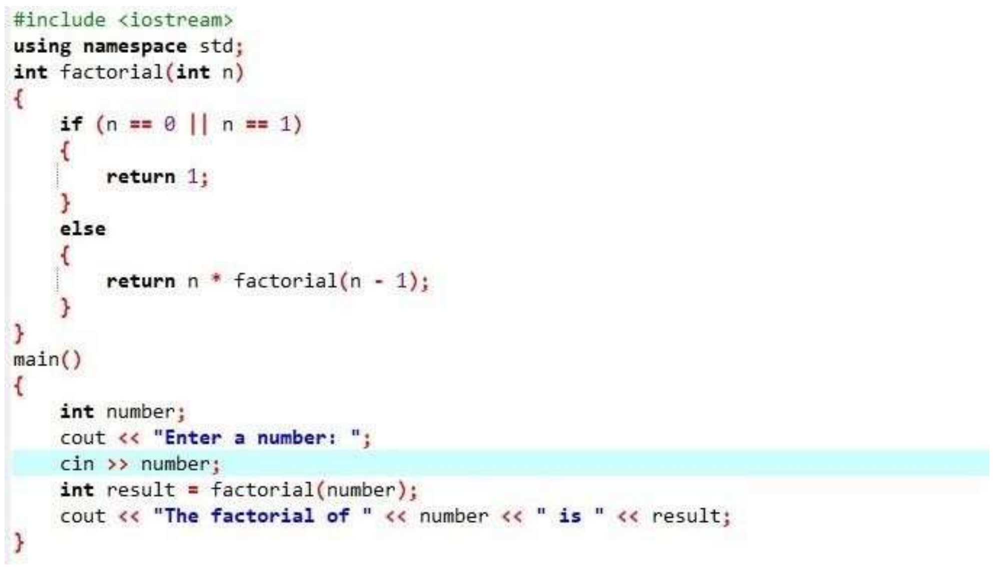
- **Function Calling**: Calling a function means invoking it in your program to perform the task it's designed for **Syntax:**

```
function_name(arguments);
```

- **What is the scope of variables in C++? Differentiate between local and global scope.**

Feature	Local Scope	Global Scope
Declaration	Inside a function, block, or loop.	Outside any function, generally at the top of the file.
Visibility	Accessible only within the block/function where declared.	Accessible throughout the entire program.
Lifetime	Exists only during the execution of the block or function.	Exists for the entire runtime of the program.
Access	Cannot be accessed by other functions.	Can be accessed by any function in the program.
Memory	Stored in the stack memory.	Stored in the data segment of the memory.

- **Explain recursion in C++ with an example.**
  - **What is Recursion in C++?** Recursion in C++ is a programming technique where a function calls itself to solve smaller instances of the same problem. In simpler terms, a recursive function is a function that solves a problem by breaking it down into smaller subproblems, and each subproblem is solved by the function itself.



```
#include <iostream>
using namespace std;
int factorial(int n)
{
    if (n == 0 || n == 1)
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}
main()
{
    int number;
    cout << "Enter a number: ";
    cin >> number;
    int result = factorial(number);
    cout << "The factorial of " << number << " is " << result;
}
```

- **What are function prototypes in C++? Why are they used?**
  - **What is a Function Prototype in C++?**
    - A **function prototype** in C++ is a declaration of a function that provides the function's name, return type, and parameters (if any), but does not include the body of the function. It tells the compiler about the function's interface before the function is actually defined.
  - **Why are Function Prototypes Used?**
    - Enables Function Calls Before Definition:
    - Helps with Type Checking:
    - Improves Code Organization:
    - Avoids Redundancy:

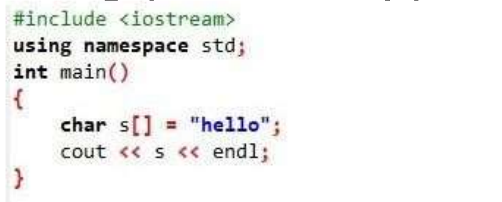
5. Arrays and Strings :

THEORY EXERCISE:

- **What are arrays in C++? Explain the difference between single dimensional and multi- dimensional arrays.**
  - **What are Arrays in C++?**
    - In C++, an **array** is a collection of variables of the same type stored in contiguous memory locations. Arrays allow you to store multiple values of the same type under a single name and access them using an index or subscript.
  - **Differences Between Single-Dimensional and Multi-Dimensional Arrays:**

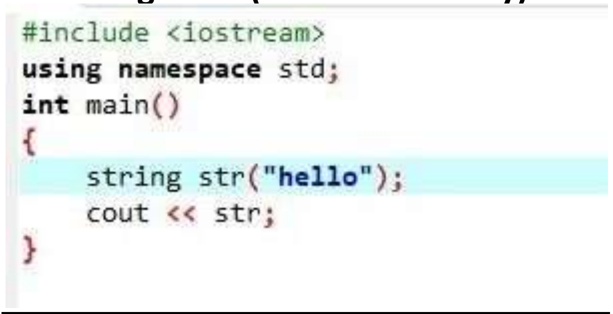
Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	A simple list of elements (linear).	An array of arrays (matrix, table, etc.).
Access Method	Accessed with one index (e.g., arr[0]).	Accessed with multiple indices (e.g., arr[0][0] for 2D).
Usage	Used for storing simple lists of data.	Used for storing tables, matrices, or complex data structures.
Dimensionality	1 dimension.	More than 1 dimension (e.g., 2D, 3D).

- **Explain string handling in C++ with examples.**
  - **String Handling in C++ :** In C++, **strings** are sequences of characters used to represent text. C++ provides two primary ways of handling strings:
    - ✓ **C-strings** (null-terminated character arrays)
    - ✓ **C++ string class** (part of the Standard Library)
  - **C-strings (Character Arrays) :**



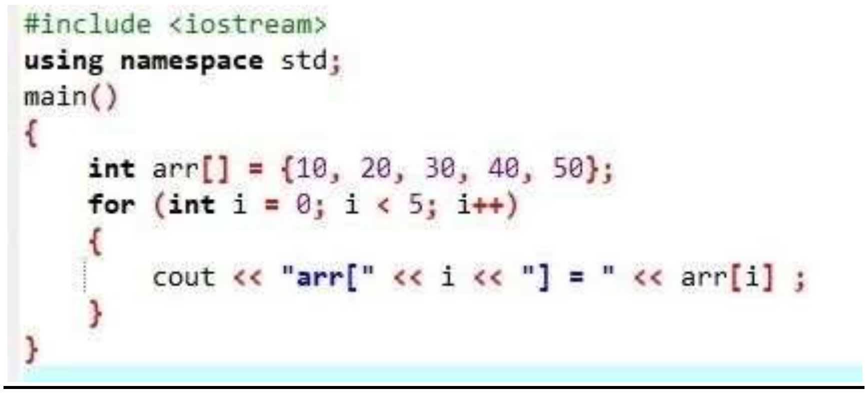
```
#include <iostream>
using namespace std;
int main()
{
    char s[] = "hello";
    cout << s << endl;
}
```

  - **C++ string Class (Standard Library) :**



```
#include <iostream>
using namespace std;
int main()
{
    string str("hello");
    cout << str;
}
```

- **How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.**
  - **Initializing 1D Arrays in C++ :**
    - <u>**Syntax :**</u> type array_name[size] = {value1, value2, value3, ...}; <u>**Example :**</u>



```
#include <iostream>
using namespace std;
main()
{
    int arr[] = {10, 20, 30, 40, 50};
    for (int i = 0; i < 5; i++)
    {
        cout << "arr[" << i << "] = " << arr[i] ;
    }
}
```

- **Initializing 2D Arrays in C++ :**
  - **<u>Syntax :</u>**

    type array_name[rows][columns]

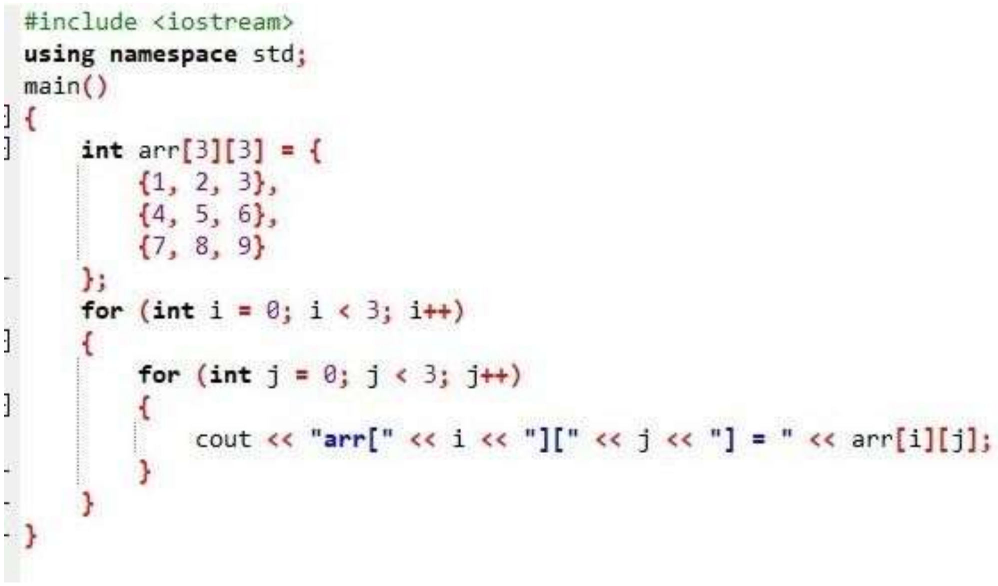
    {

    {value1, value2, ..., valueN},

    {value1, value2, ..., valueN},

    };

  - **Example :**



```
#include <iostream>
using namespace std;
main()
{
    int arr[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << "arr[" << i << "][" << j << "] = " << arr[i][j];
        }
    }
}
```

6. Introduction to Object-Oriented Programming :

THEORY EXERCISE:

- **Explain the key concepts of Object-Oriented Programming (OOP).**

    - **Class :** A class is a blueprint or template for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from the class will have.

    - **Object :** An object is an instance of a class. Once a class is defined, objects can be created (instantiated) from it. Each object has its own distinct values for the class attributes.

    - **Encapsulation :** Encapsulation is the concept of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class.

    - **Inheritance :** Inheritance allows a class to inherit properties and behaviors (methods) from another class. This promotes code reusability and establishes a hierarchical relationship between classes.

    - **Abstraction :** Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. It helps to reduce complexity and increase efficiency.

    - **Composition :** Composition is a special form of association where one object contains another object as part of its structure. It implies a "has-a" relationship, meaning the contained object is an essential part of the container object.

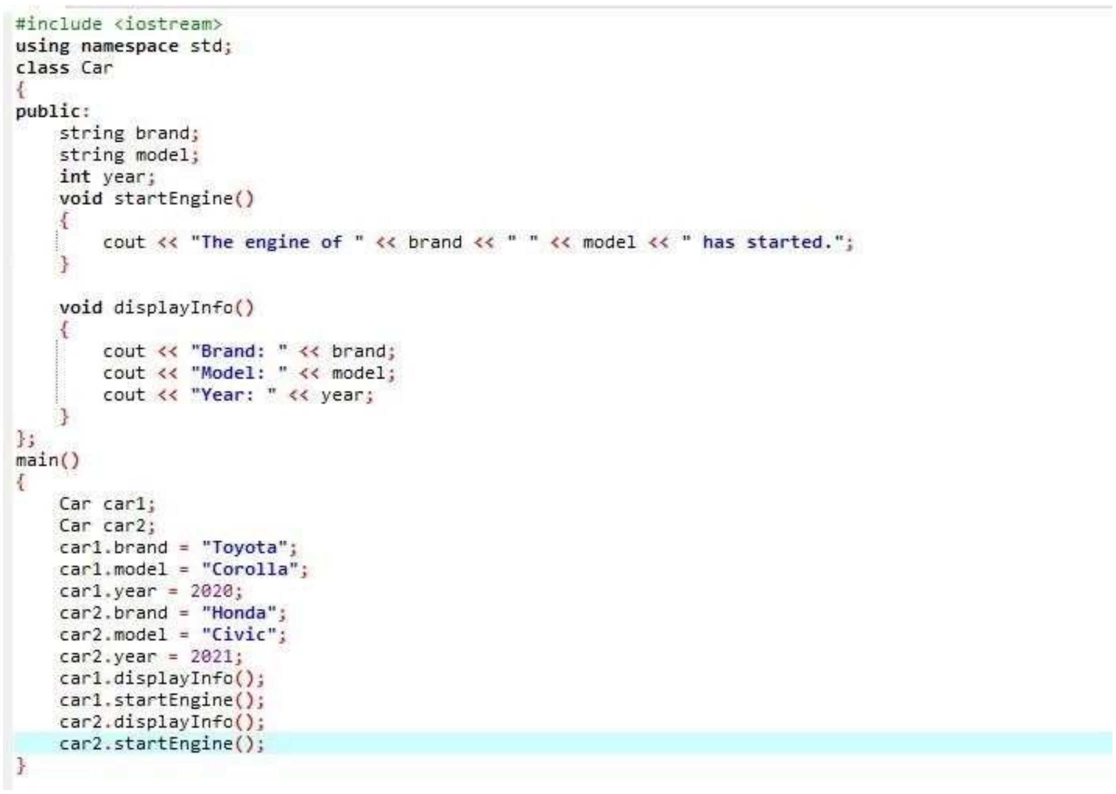
    - **Aggregation :** Aggregation is a form of association where one object can contain another, but the contained object can exist independently of the container object.

- **What are classes and objects in C++? Provide an example.**

    - **Class :** A class in C++ is a user-defined data type that serves as a blueprint for creating objects. It defines a set of attributes (data members) and methods (functions) that operate on these attributes. Classes encapsulate data and functions together, promoting reusability, modularity, and organization in code.

    - **Object :** An object is an instance of a class. Once a class is defined, we can create multiple objects from that class. Each object will have its own copy of the data members and can invoke the methods defined in the class.

    - **Example of a class and object in C++:**



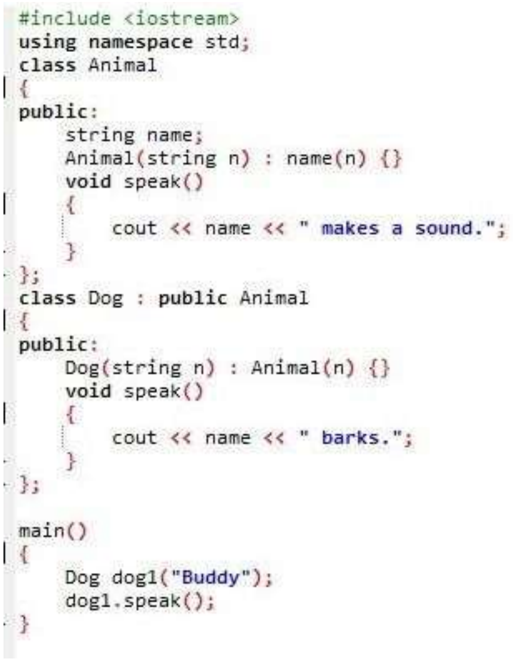
```
#include <iostream>
using namespace std;
class Car
{
public:
    string brand;
    string model;
    int year;
    void startEngine()
    {
        cout << "The engine of " << brand << " " << model << " has started.";
    }

    void displayInfo()
    {
        cout << "Brand: " << brand;
        cout << "Model: " << model;
        cout << "Year: " << year;
    }
};
main()
{
    Car car1;
    Car car2;
    car1.brand = "Toyota";
    car1.model = "Corolla";
    car1.year = 2020;
    car2.brand = "Honda";
    car2.model = "Civic";
    car2.year = 2021;
    car1.displayInfo();
    car1.startEngine();
    car2.displayInfo();
    car2.startEngine();
}
```

- **What is inheritance in C++? Explain with an example.**

  - **Inheritance in C++ :** Inheritance is a fundamental concept in Object-Oriented Programming (OOP), including in C++. It allows a class (called the **derived class**) to inherit attributes and methods from another class (called the **base class**). This enables **code reusability** and creates a hierarchical relationship between the base and derived classes.

  - **Example of Inheritance in C++ :**



```
#include <iostream>
using namespace std;
class Animal
{
public:
    string name;
    Animal(string n) : name(n) {}
    void speak()
    {
        cout << name << " makes a sound.";
    }
};
class Dog : public Animal
{
public:
    Dog(string n) : Animal(n) {}
    void speak()
    {
        cout << name << " barks.";
    }
};

main()
{
    Dog dog1("Buddy");
    dog1.speak();
}
```

- **What is encapsulation in C++? How is it achieved in classes?**
  - **Encapsulation in C++ :** Encapsulation is one of the core principles of Object-
    Oriented Programming (OOP). It refers to the concept of **bundling** data (attributes) and the methods (functions) that operate on that data into a **single unit**, typically a class. Encapsulation also involves **restricting direct access** to some of an object's components, which is achieved by making some attributes and methods **private** or **protected** and providing **public** methods to access and modify them.

  - **Achieving Encapsulation in C++ :**
    - ✓ **public**: Members (attributes or methods) declared as public can be accessed from outside the class.
    - ✓ **private**: Members declared as private are accessible only within the class itself.

    They cannot be accessed directly from outside the class.
    - ✓ **protected**: Members declared as protected are accessible within the class and by derived classes, but not by objects or classes outside the hierarchy.