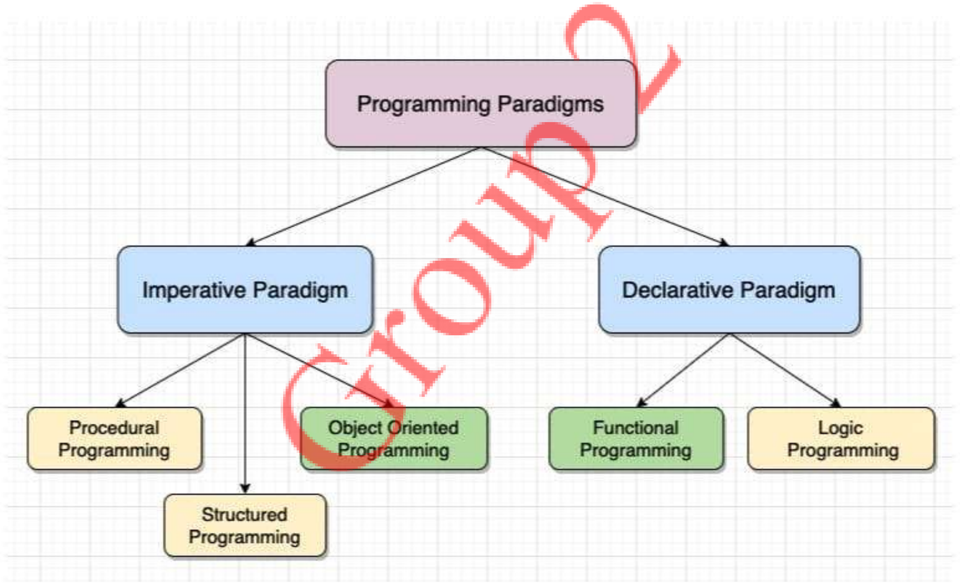
"Introduction to Object-Oriented Programming Concepts in C++ with Examples"

Introduction to Programming Paradigms

Definition: A programming paradigm refers to a fundamental style or methodology of computer programming. It is a framework that defines how programmers write, organize, and structure code to solve problems. Each paradigm provides a distinct approach to designing and implementing software, focusing on different ways to represent data, perform computations, and manage program behavior**.**

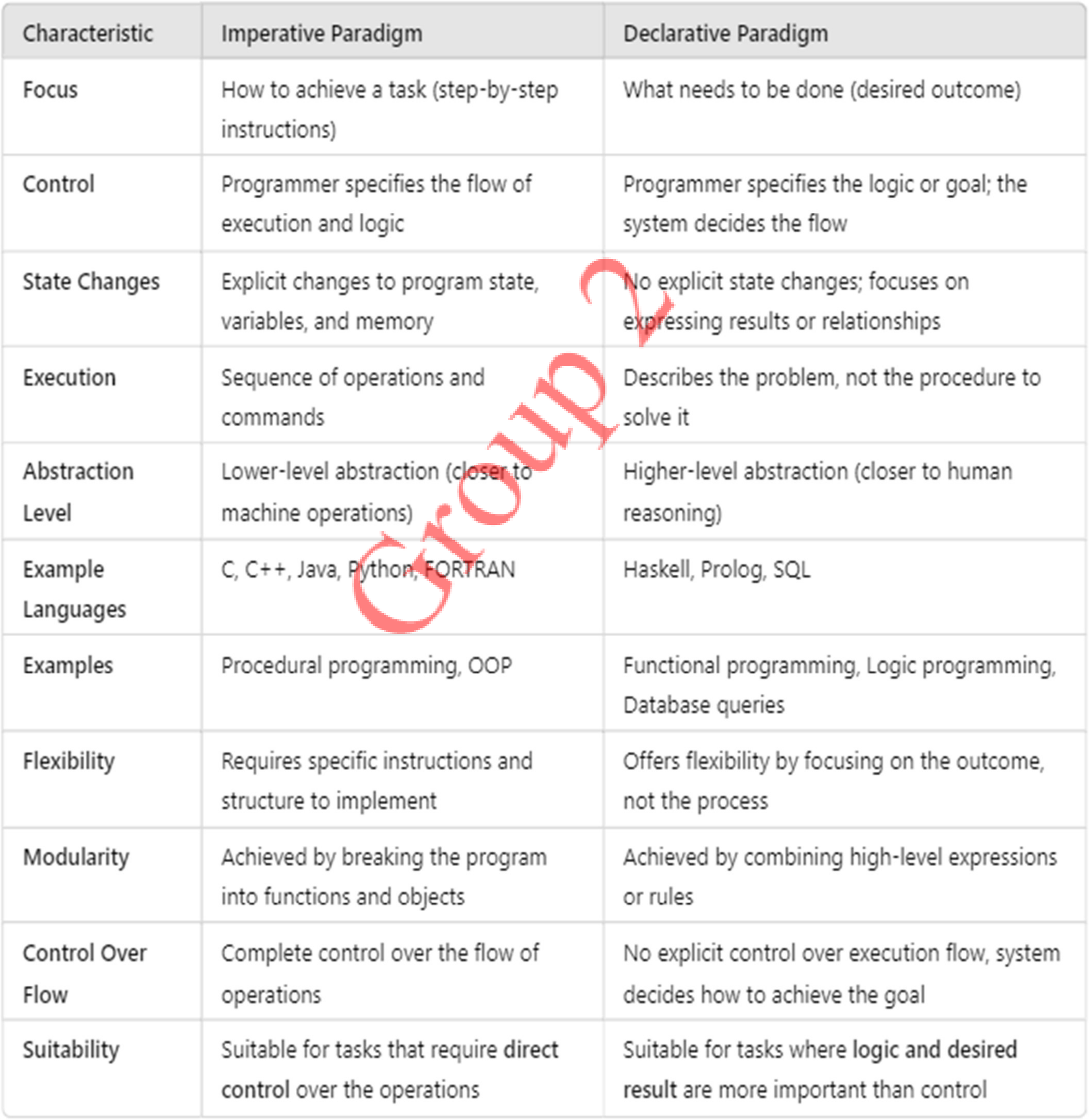


Imperative Paradigm

The Imperative Paradigm focuses on how a task is performed by specifying the sequence of commands or instructions that change the program's state. In this paradigm, the programmer describes the steps or operations that must be performed to achieve the desired result.

<u>Declarative Paradigm</u>

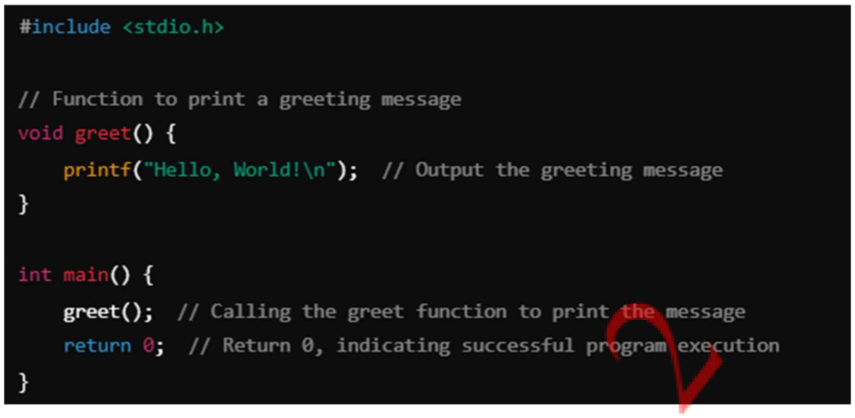
The Declarative Paradigm focuses on what needs to be done, without specifying the exact steps or sequence of operations to achieve the result. In this paradigm, the programmer expresses the desired outcome, and the system decides how to achieve it.



Characteristic	Imperative Paradigm	Declarative Paradigm
Focus	How to achieve a task (step-by-step instructions)	What needs to be done (desired outcome)
Control	Programmer specifies the flow of execution and logic	Programmer specifies the logic or goal; the system decides the flow
State Changes	Explicit changes to program state, variables, and memory	No explicit state changes; focuses on expressing results or relationships
Execution	Sequence of operations and commands	Describes the problem, not the procedure to solve it
Abstraction Level	Lower-level abstraction (closer to machine operations)	Higher-level abstraction (closer to human reasoning)
Example Languages	C, C++, Java, Python, FORTRAN	Haskell, Prolog, SQL
Examples	Procedural programming, OOP	Functional programming, Logic programming, Database queries
Flexibility	Requires specific instructions and structure to implement	Offers flexibility by focusing on the outcome, not the process
Modularity	Achieved by breaking the program into functions and objects	Achieved by combining high-level expressions or rules
Control Over Flow	Complete control over the flow of operations	No explicit control over execution flow, system decides how to achieve the goal
Suitability	Suitable for tasks that require **direct control** over the operations	Suitable for tasks where **logic and desired result** are more important than control

1. **Procedural Programming:** Procedural Programming is a programming paradigm derived from the imperative paradigm. In procedural programming, a program is organized as a sequence of instructions or procedures (also known as functions or routines) that perform specific tasks.

   Examples: C, Pascal, FORTRAN.

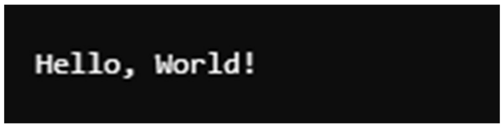


```c
#include <stdio.h>

// Function to print a greeting message
void greet() {
    printf("Hello, World!\n");  // Output the greeting message
}

int main() {
    greet();  // Calling the greet function to print the message
    return 0;  // Return 0, indicating successful program execution
}
```

Explanation:

1. #include <stdio.h>: This header is used in C for input and output operations, particularly for functions like printf().
2. void greet() Function: This function simply prints "Hello, World!" to the console using printf().
3. main() Function: This is the entry point of the program. It calls the greet() function and then returns 0 to indicate successful execution.

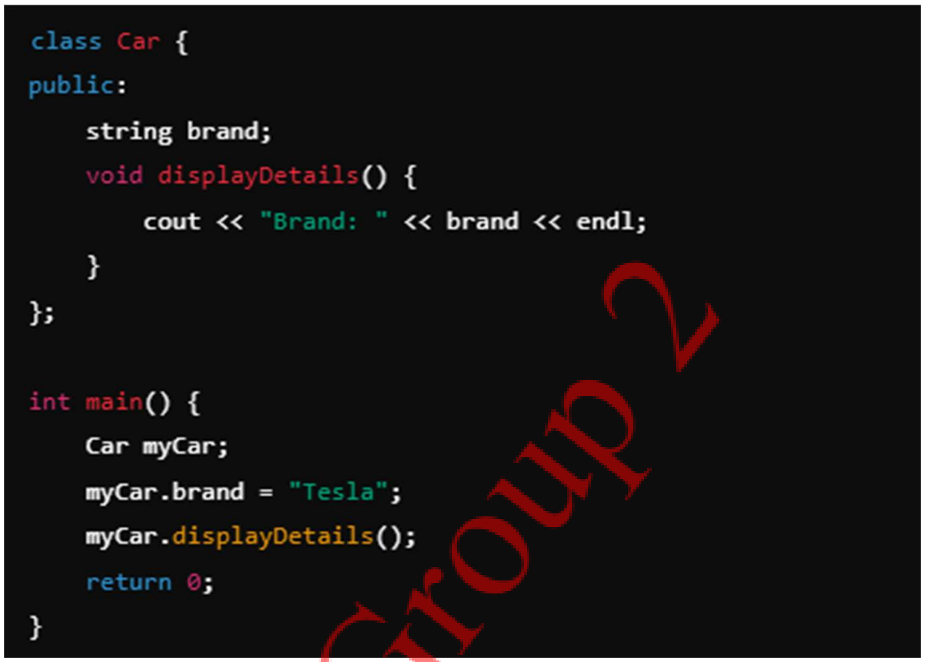
Output:



```
Hello, World!
```

2. **Object-Oriented Programming (OOP):** Object-Oriented Programming (OOP) is an extension of the imperative paradigm, focusing on objects (instances of classes). It introduces features like encapsulation, inheritance, and polymorphism to better structure programs.

   Examples: C++, JAVA, Python, Ruby, C#, PHP etc.

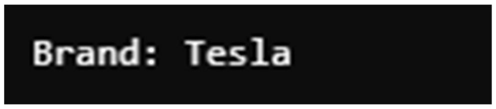


```
class Car {
public:
    string brand;
    void displayDetails() {
        cout << "Brand: " << brand << endl;
    }
};

int main() {
    Car myCar;
    myCar.brand = "Tesla";
    myCar.displayDetails();
    return 0;
}
```

Explanation:

1. A Car class is defined with a data member brand and a member function displayDetails() to display the car's brand.
2. In the main() function, an object myCar of class Car is created.
3. The brand of myCar is assigned the value "Tesla".
4. The displayDetails() function is called to print the brand of the car.
5. The code demonstrates object creation, data assignment, and method invocation in OOP.

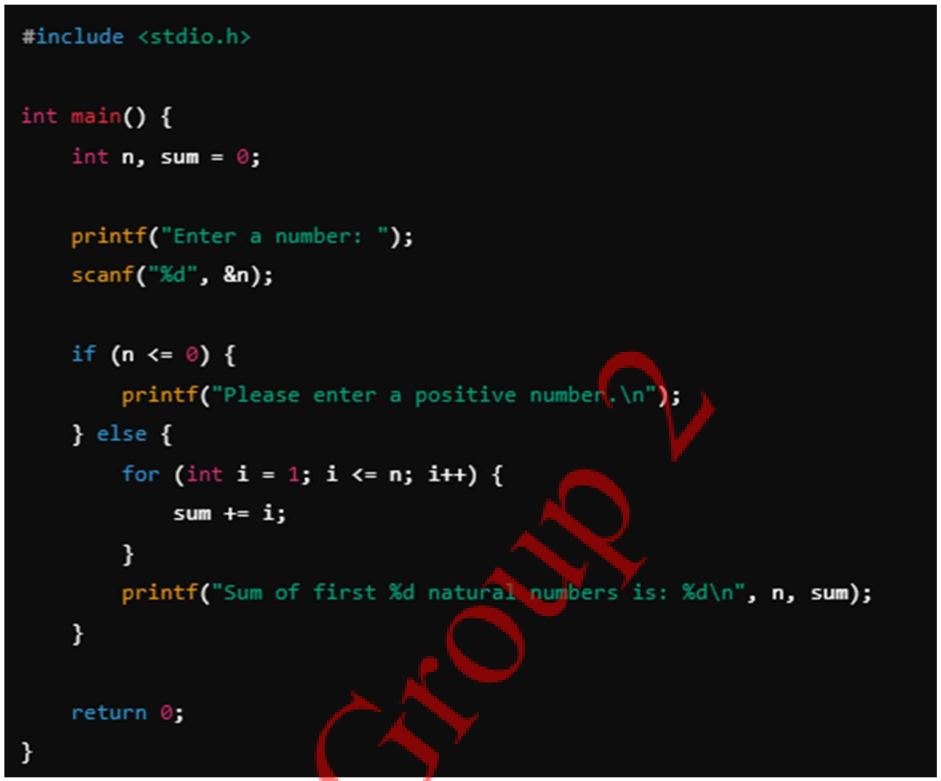
Output:



```
Brand: Tesla
```

3. **Structured programming:** Structured programming is a programming paradigm that emphasizes breaking down a program into smaller, manageable sections or blocks, typically using control structures like loops, conditionals, and functions.

   Examples: C, Pascal, FORTR



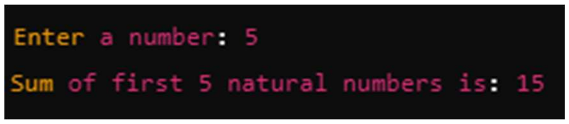
```c
#include <stdio.h>

int main() {
    int n, sum = 0;

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

    if (n <= 0) {
        printf("Please enter a positive number.\n");
    } else {
        for (int i = 1; i <= n; i++) {
            sum += i;
        }
        printf("Sum of first %d natural numbers is: %d\n", n, sum);
    }

    return 0;
}
```

Explanation:

1. The program asks the user to input a number n.
2. It checks if the entered number is positive. If not, it prompts the user to enter a positive number.
3. If the number is valid, it calculates the sum of the first n natural numbers using a for loop and prints calculated sum.
4. The program ensures structured flow with input, validation, iteration, and output.

AN, BASIC, JavaScript etc.

Output:

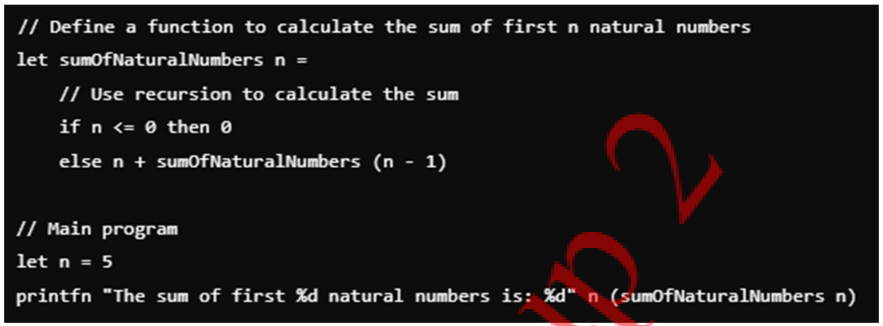


```
Enter a number: 5
Sum of first 5 natural numbers is: 15
```

Types of Declarative Programming:

1. **Functional programming:** Functional programming (FP) is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It focuses on using functions as the primary building blocks of programs rather than relying on objects or procedural code.

   Example: Haskell, Lisp (and its dialects like Clojure, Scheme), F# etc.

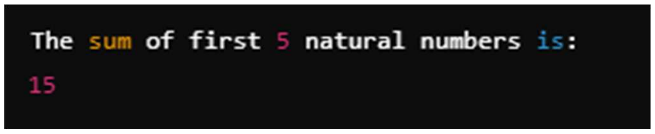


```
// Define a function to calculate the sum of first n natural numbers
let sumOfNaturalNumbers n =
    // Use recursion to calculate the sum
    if n <= 0 then 0
    else n + sumOfNaturalNumbers (n - 1)

// Main program
let n = 5
printfn "The sum of first %d natural numbers is: %d" n (sumOfNaturalNumbers n)
```

Explanation:

1. **sumOfNaturalNumbers** is a recursive function that calculates the sum of the first n natural numbers.
2. The function checks if n is 0 or negative, returning 0 in that case.
3. Otherwise, it adds n to the sum of the remaining numbers using recursion.
4. In the main part of the program, it prints the sum of the first 5 natural numbers.

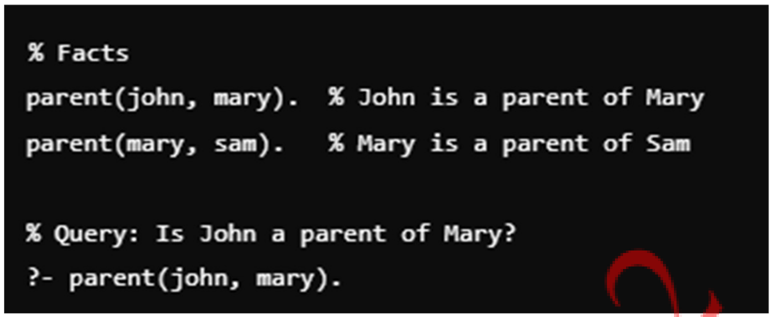
Output:



```
The sum of first 5 natural numbers is:
15
```

2. **Logical Programing:** Logical programming (or Logic programming) is a programming paradigm that is based on formal logic. In logic programming, a program is written as a set of logical statements (called facts and rules), and the computer uses a deductive reasoning process to infer new facts and answer queries based on the given facts and rules.

   Example: Prolog, LISP, Datalog.



```
% Facts
parent(john, mary).  % John is a parent of Mary
parent(mary, sam).   % Mary is a parent of Sam

% Query: Is John a parent of Mary?
?- parent(john, mary).
```

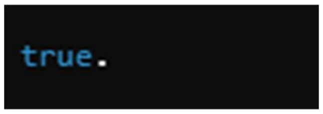
   Explanation:

1. Facts:

   - parent(john, mary).: This means John is a parent of Mary.
   - parent(mary, sam).: This means Mary is a parent of Sam.

2. **Query**:
   - ?-parent(john, mary).: This asks Prolog, "Is John a parent of Mary?"
   -

   Output:



```
true.
```

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects." Objects are self-contained units that combine data (attributes) and functions (methods) into a single entity. OOP is designed to provide a natural way of thinking about and structuring software by modeling it after real-world entities.

Why Object-Oriented Programming?

Before we discuss object-oriented programming, we need to learn why we need object-oriented programming?

- C++ language was designed with the main intention of adding object-oriented programming to C language
- As the size of the program increases readability, maintainability, and bug-free nature of the program decrease.
- This was the major problem with languages like C which relied upon functions or procedure (hence the name procedural programming language)
- As a result, the possibility of not addressing the problem adequately was high
- Also, data was almost neglected, data security was easily compromised
- Using classes solves this problem by modeling program as a real-world scenario

Difference between Procedure Oriented Programming and Object-Oriented Programming

Procedure Oriented Programming

- Consists of writing a set of instruction for the computer to follow
- The main focus is on functions and not on the flow of data
- Functions can either use local or global data
- Data moves openly from function to function

Object-Oriented Programming

- Works on the concept of classes and object

- A class is a template to create objects
- Treats data as a critical element
- Decomposes the problem in objects and builds data and functions around the objects

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. The primary characteristics or principles of OOP include:

1. Encapsulation

- Encapsulation involves bundling data (attributes) and methods (functions) that operate on the data into a single unit, called an object.
- It restricts direct access to some of the object's components, maintaining control and security by exposing only necessary information through methods (getters/setters).
- Example: Using private variables in a class and accessing them via public methods.

---

2. Abstraction

- Abstraction focuses on exposing only essential details to the user while hiding the implementation complexity.
- It simplifies programming by reducing complexity and allowing the programmer to focus on interactions with objects.
- Example: A "Car" object abstracts away internal details like the engine but provides functionalities like "start" and "drive."

---

3. Inheritance

- Inheritance allows one class (child class) to acquire properties and methods of another class (parent class).
- It promotes code reuse and establishes a relationship between classes.
- Example: A "Dog" class can inherit from an "Animal" class, reusing general animal properties and behaviors.

---

4. Polymorphism

- Polymorphism means "many forms." It allows objects to be treated as instances of their parent class rather than their actual class.
- It enables methods to perform differently based on the object they are called on or the arguments passed.
- Example:
  - Method Overloading: Multiple methods with the same name but different parameters.
  - Method Overriding: A child class provides a specific implementation of a method already defined in its parent class.

---

5. Class and Object

- **Class**: A blueprint for creating objects. It defines the structure and behavior of an object.
- **Object**: An instance of a class that encapsulates data and methods.

---

6. Modularity

- OOP promotes modularity by breaking down a program into smaller, manageable, and reusable parts (classes and objects).
- This makes the code more maintainable and easier to debug.

---

7. Dynamic Binding

- Decisions regarding method execution are made at runtime, not at compile time.
- Example: A parent class reference can point to a child class object, and the appropriate method is called based on the actual object type.

---

8. Message Passing

- Objects communicate with each other via messages (method calls). Each object contains data and methods to interact with other objects.

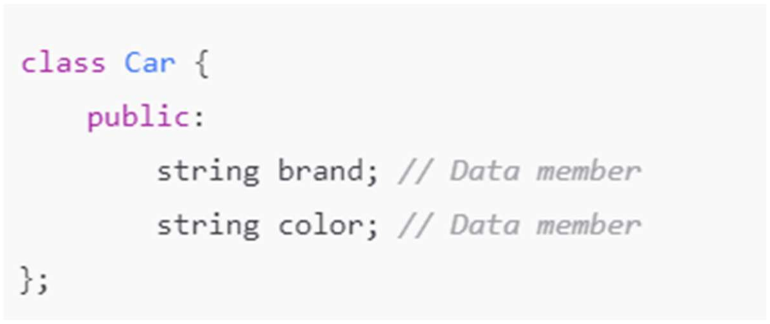
➔In short to understand we can take reference from the following :-

Basic Concepts in Object-Oriented Programming

- **Classes -** Basic template for creating objects
- **Objects –** Basic run-time entities
- **Data Abstraction & Encapsulation –** Wrapping data and functions into a single unit
- **Inheritance –** Properties of one class can be inherited into others
- **Polymorphism –** Ability to take more than one forms
- **Dynamic Binding –** Code which will execute is not known until the program runs
- **Message Passing –** message (Information) call format

Data Member

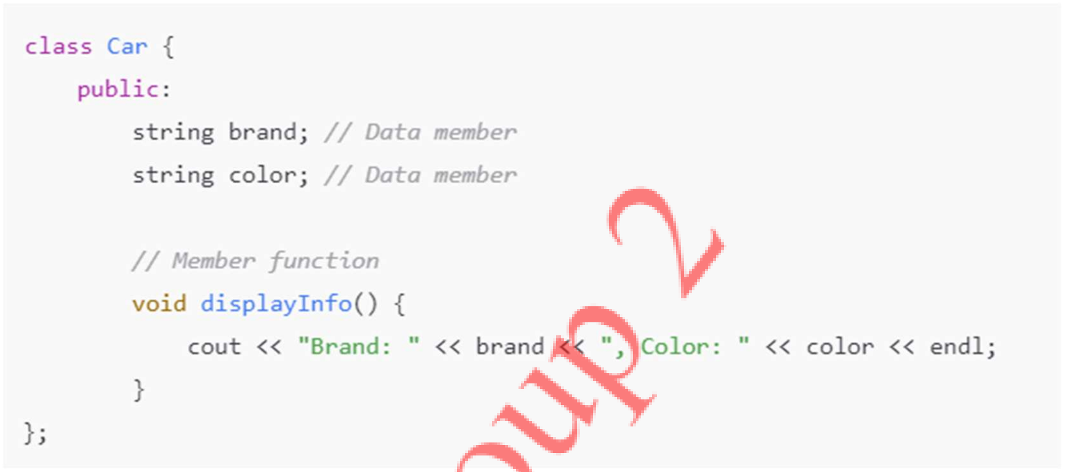
- **Definition**: A **data member** refers to the variables or attributes defined inside a class. These variables hold the data or state of an object created from the class.
- **Purpose**: They define the properties of a class or object.
- **Scope**: They are usually private (or protected) to ensure encapsulation, and access to them is provided through getter and setter methods.



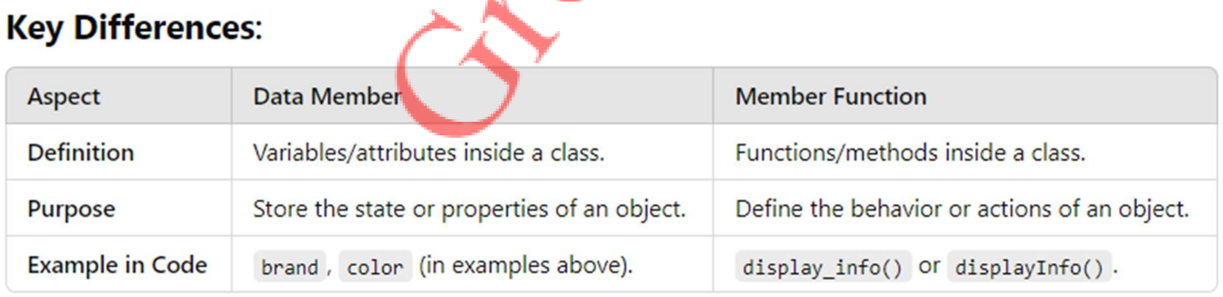
```
class Car {
    public:
        string brand; // Data member
        string color; // Data member
};
```

Member Function

- **Definition**: A **member function** refers to the methods or functions defined inside a class that operate on the data members of that class.
- **Purpose**: They define the behavior or actions that an object of the class can perform.
- **Scope**: They can be public, private, or protected depending on how they should be accessed.



```
class Car {
    public:
        string brand; // Data member
        string color; // Data member

        // Member function
        void displayInfo() {
            cout << "Brand: " << brand << ", Color: " << color << endl;
        }
};
```



Key Differences:

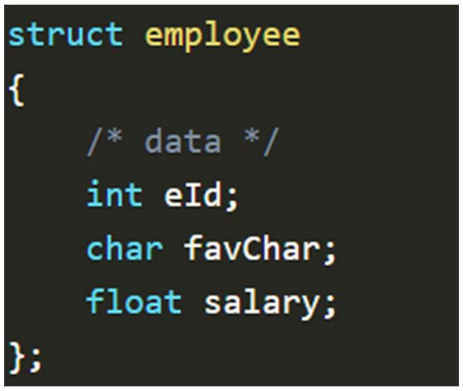
Aspect	Data Member	Member Function
Definition	Variables/attributes inside a class.	Functions/methods inside a class.
Purpose	Store the state or properties of an object.	Define the behavior or actions of an object.
Example in Code	`brand`, `color` (in examples above).	`display_info()` or `displayInfo()`.

Structures in C++

The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.

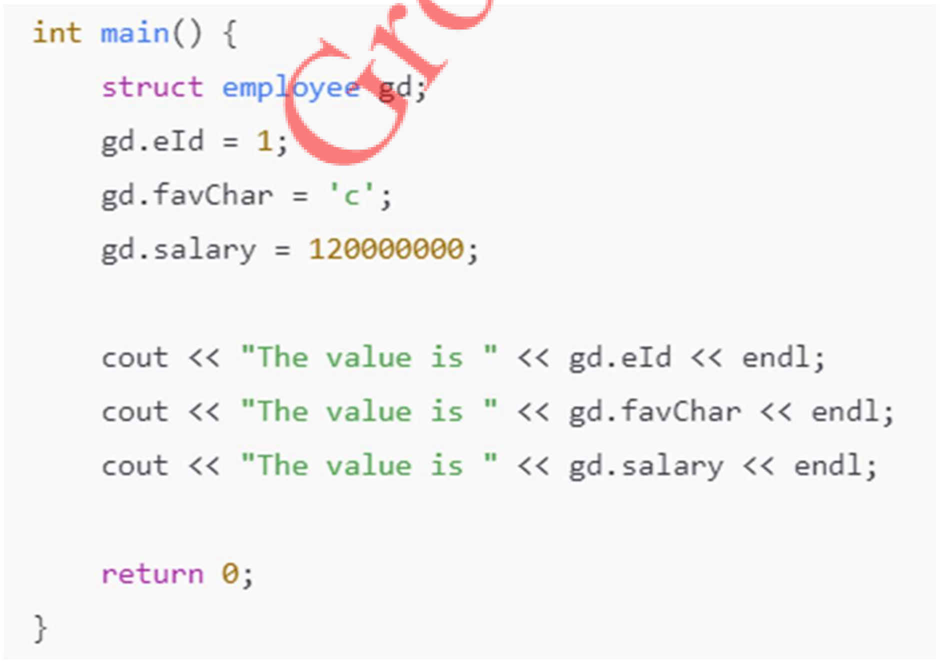
Example:

As shown in Fig 1, we have created a structure with the name “employee”, in which we have declared three variables of different data types (eId, favchar, salary).



```
struct employee
{
    /* data */
    int eId;
    char favChar;
    float salary;
};
```

Fig 1

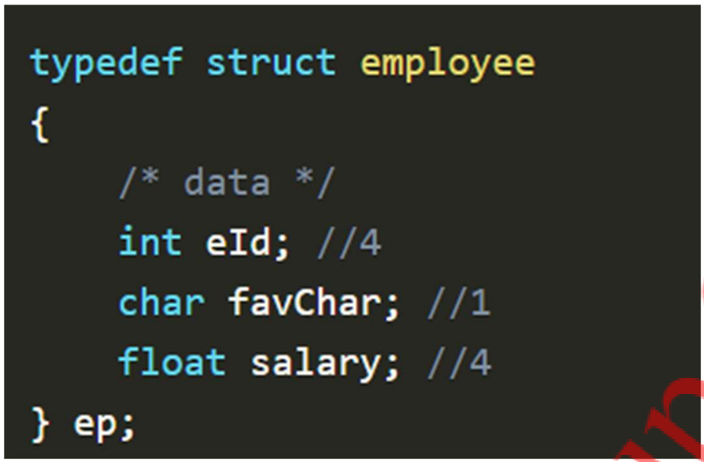


```
int main() {
    struct employee gd;
    gd.eId = 1;
    gd.favChar = 'c';
    gd.salary = 120000000;

    cout << "The value is " << gd.eId << endl;
    cout << "The value is " << gd.favChar << endl;
    cout << "The value is " << gd.salary << endl;

    return 0;
}
```

Fig 2

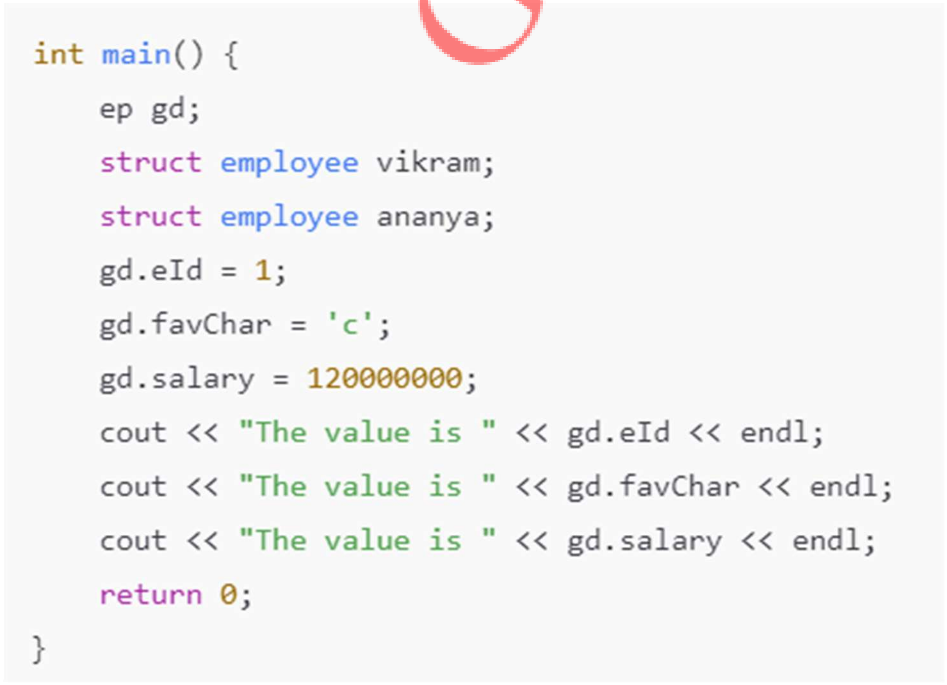
As shown in Code Snippet 2, 1st we have created a structure variable "harry" of type "employee", 2nd we have assigned values to (eId, favchar, salary) fields of the structure employee and at the end we have printed the value of "salary".

Another way to create structure variables without using the keyword "struct" and the name of the struct is using typedef.



```
typedef struct employee
{
    /* data */
    int eId; //4
    char favChar; //1
    float salary; //4
} ep;
```

we have used a keyword "**typedef**" before struct and after the closing bracket of structure, we have written "ep". Now we can create structure variables without using the keyword "struct" and name of the struct.

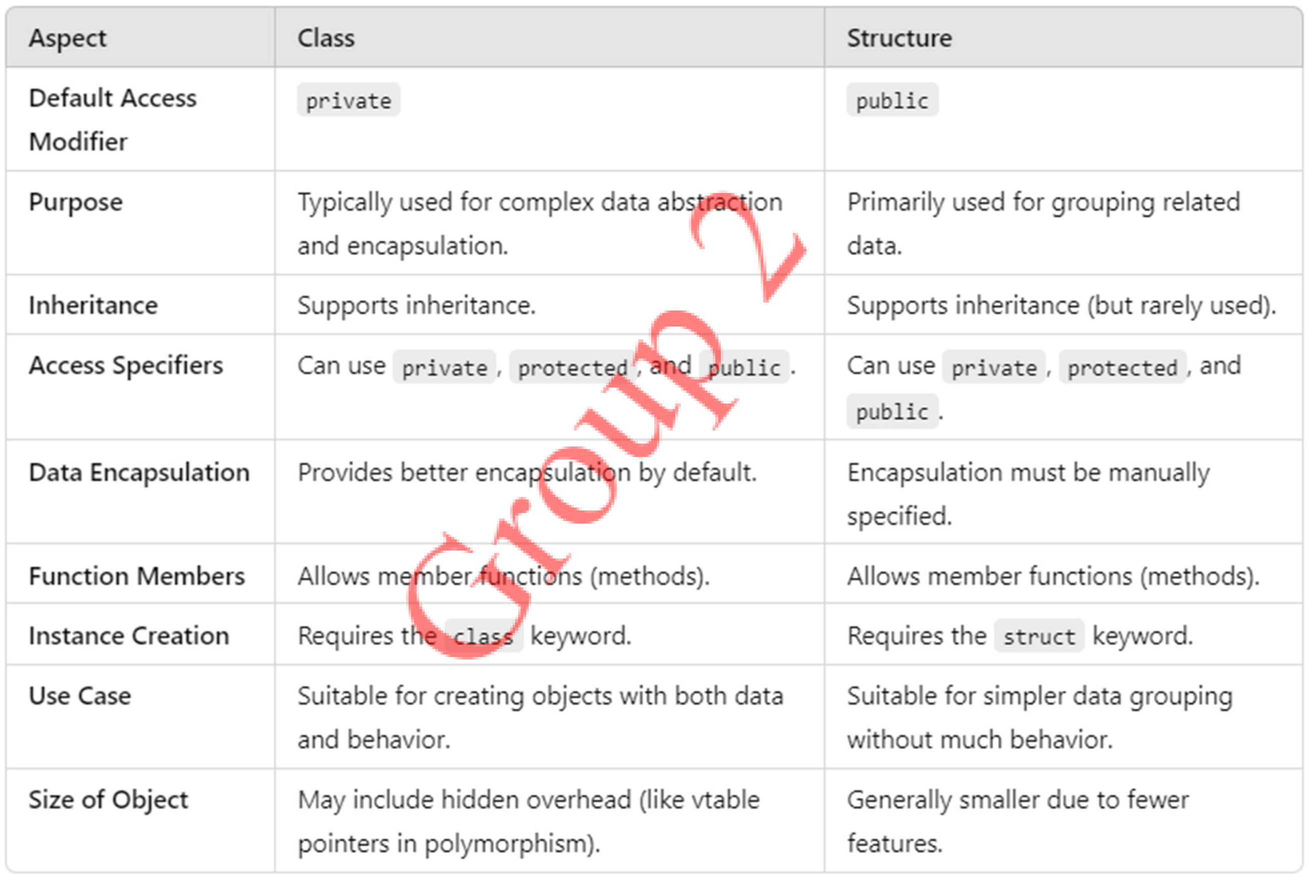


```
int main() {
    ep gd;
    struct employee vikram;
    struct employee ananya;
    gd.eId = 1;
    gd.favChar = 'c';
    gd.salary = 120000000;
    cout << "The value is " << gd.eId << endl;
    cout << "The value is " << gd.favChar << endl;
    cout << "The value is " << gd.salary << endl;
    return 0;
}
```

Difference Between class and Structures:

Why use classes instead of structures

Classes and structures are somewhat the same but still, they have some differences. For example, we cannot hide data in structures which means that everything is public and can be accessed easily which is a major drawback of the structure because structures cannot be used where data security is a major concern. Another drawback of structures is that we cannot add functions in it.



Aspect	Class	Structure
Default Access Modifier	`private`	`public`
Purpose	Typically used for complex data abstraction and encapsulation.	Primarily used for grouping related data.
Inheritance	Supports inheritance.	Supports inheritance (but rarely used).
Access Specifiers	Can use `private`, `protected`, and `public`.	Can use `private`, `protected`, and `public`.
Data Encapsulation	Provides better encapsulation by default.	Encapsulation must be manually specified.
Function Members	Allows member functions (methods).	Allows member functions (methods).
Instance Creation	Requires the `class` keyword.	Requires the `struct` keyword.
Use Case	Suitable for creating objects with both data and behavior.	Suitable for simpler data grouping without much behavior.
Size of Object	May include hidden overhead (like vtable pointers in polymorphism).	Generally smaller due to fewer features.

Classes, Public and Private access modifiers in C++

Classes in C++

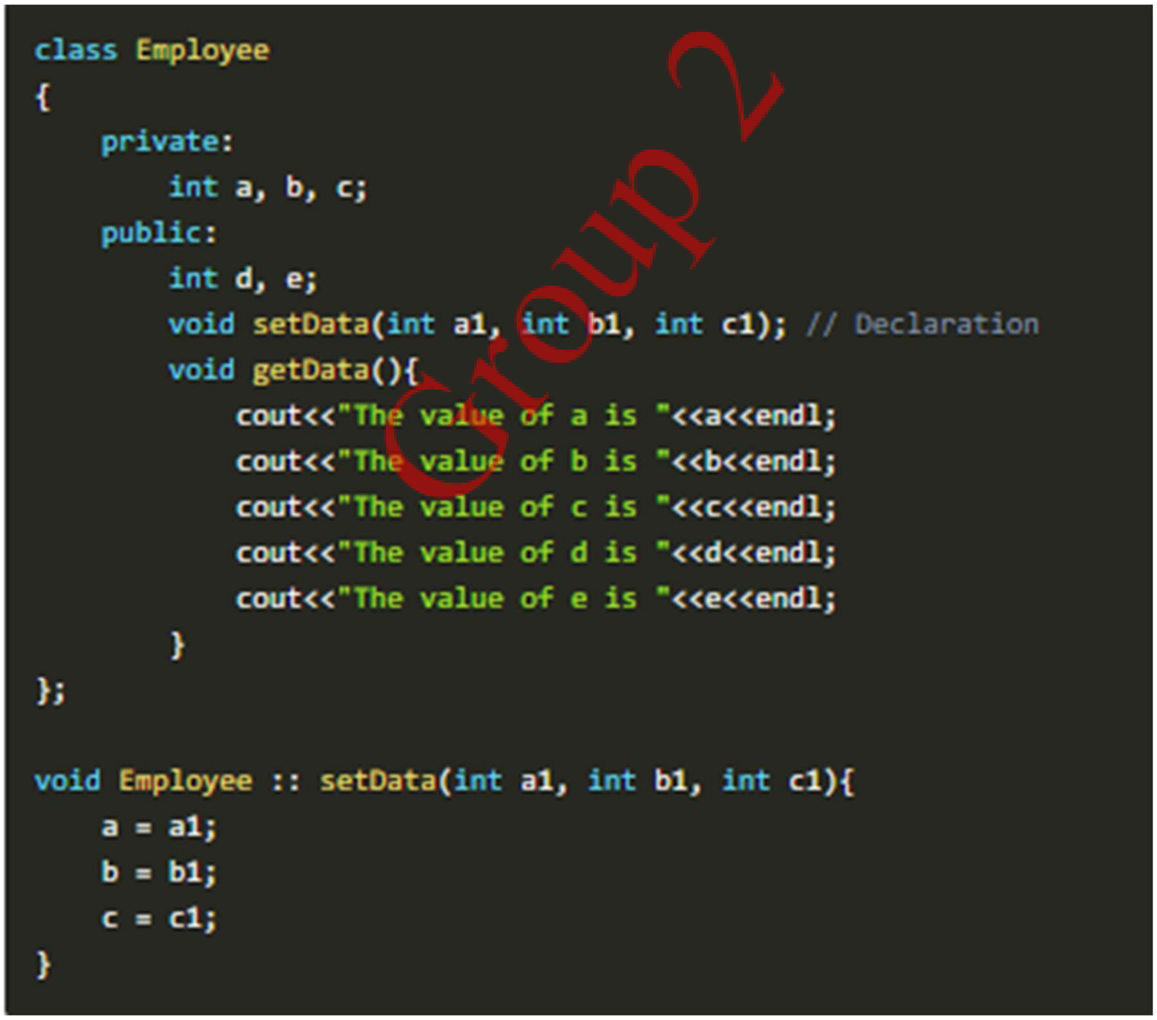
Classes are user-defined data-types and are a template for creating objects. Classes consist of variables and functions which are also called class members.

Public Access Modifier in C++

All the variables and functions declared under public access modifier will be available for everyone. They can be accessed both inside and outside the class. Dot (.) operator is used in the program to access public data members directly.

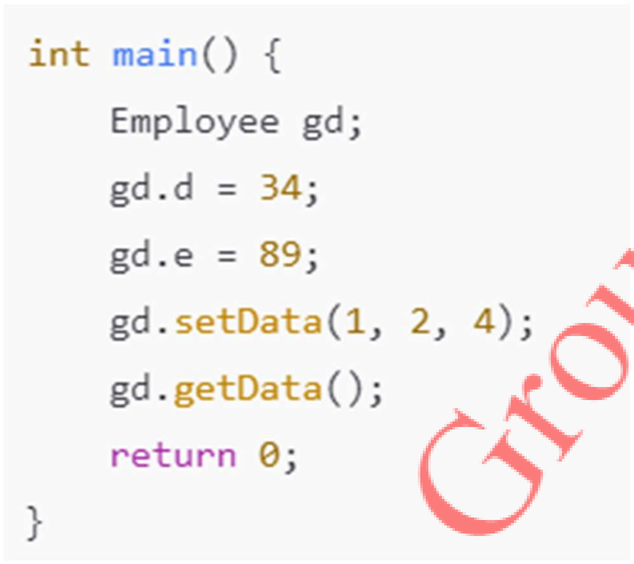
Private Access Modifier in C++

All the variables and functions declared under a private access modifier can only be used inside the class. They are not permissible to be used by any object or function outside the class.

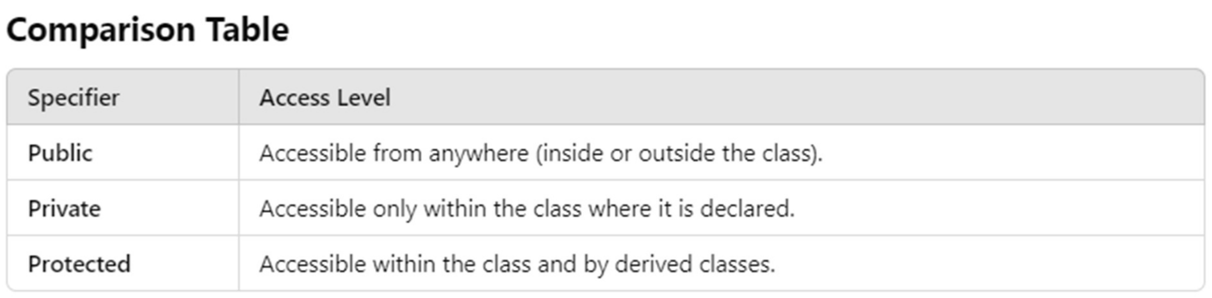


```
class Employee
{
    private:
        int a, b, c;
    public:
        int d, e;
        void setData(int a1, int b1, int c1); // Declaration
        void getData(){
            cout<<"The value of a is "<<a<<endl;
            cout<<"The value of b is "<<b<<endl;
            cout<<"The value of c is "<<c<<endl;
            cout<<"The value of d is "<<d<<endl;
            cout<<"The value of e is "<<e<<endl;
        }
};

void Employee :: setData(int a1, int b1, int c1){
    a = a1;
    b = b1;
    c = c1;
}
```

As shown in above screenshot, 1st we created an “employee” class, 2nd three integer variables “int a”, “int b”, and “int c” were declared under the private access modifier, 3rd two integer variables “int d” and “int e” was declared under the public access modifiers, 4th “setData” function was declared, 5th “getData” function was defined and values of all the variables are printed. 6th “setData” function was defined outside the “employee” class by using a scope resolution operator; “setData” function is used to assign values to the private member of the class. An example to create the object of the class and use its class members is shown below:-

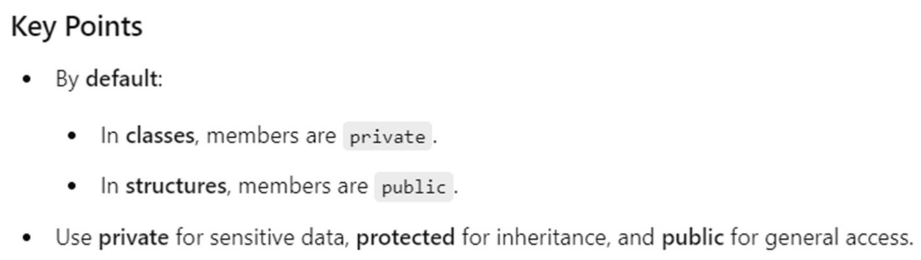


```
int main() {
    Employee gd;
    gd.d = 34;
    gd.e = 89;
    gd.setData(1, 2, 4);
    gd.getData();
    return 0;
}
```



Comparison Table

Specifier	Access Level
Public	Accessible from anywhere (inside or outside the class).
Private	Accessible only within the class where it is declared.
Protected	Accessible within the class and by derived classes.



Key Points

- By **default**:
  - In **classes**, members are `private`.
  - In **structures**, members are `public`.
- Use **private** for sensitive data, **protected** for inheritance, and **public** for general access.

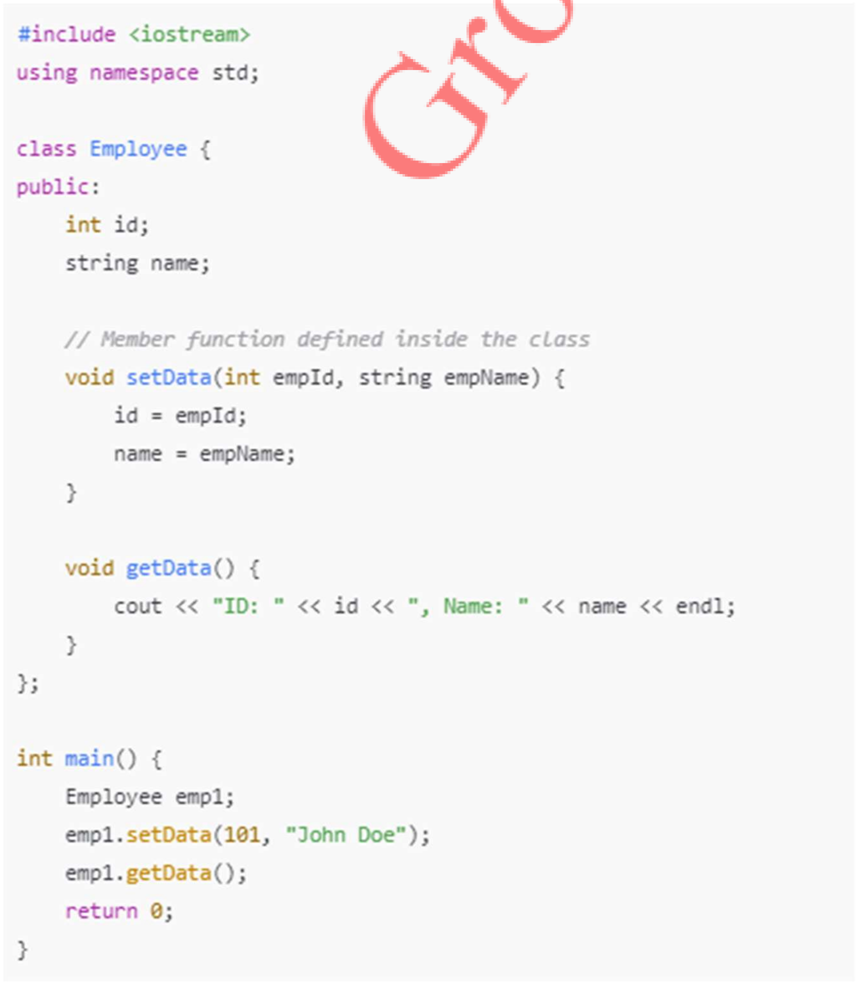
Defining Member Functions Inside and Outside the Class in C++

Definition:

In C++, member functions of a class can be defined **inside** or **outside** the class. Defining them inside the class makes the code more compact, while defining them outside the class keeps the class definition cleaner.

Member Function Defined Inside the Class:

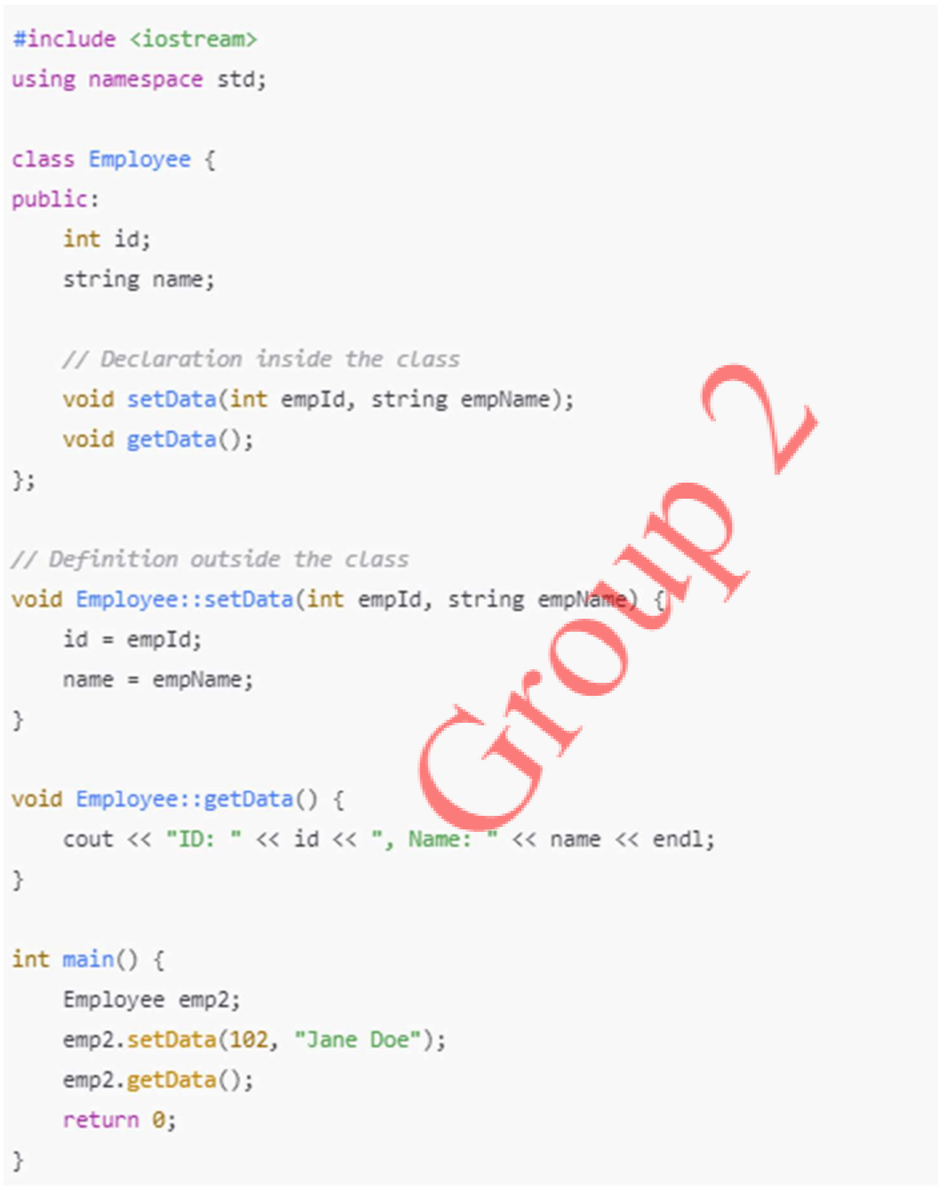
When the member function is defined inside the class, it's automatically considered inline.



```
#include <iostream>
using namespace std;

class Employee {
public:
    int id;
    string name;

    // Member function defined inside the class
    void setData(int empId, string empName) {
        id = empId;
        name = empName;
    }

    void getData() {
        cout << "ID: " << id << ", Name: " << name << endl;
    }
};

int main() {
    Employee emp1;
    emp1.setData(101, "John Doe");
    emp1.getData();
    return 0;
}
```

Member Function Defined Outside the Class:

When the member function is defined outside the class, you need to declare the function prototype inside the class, and then define it outside.

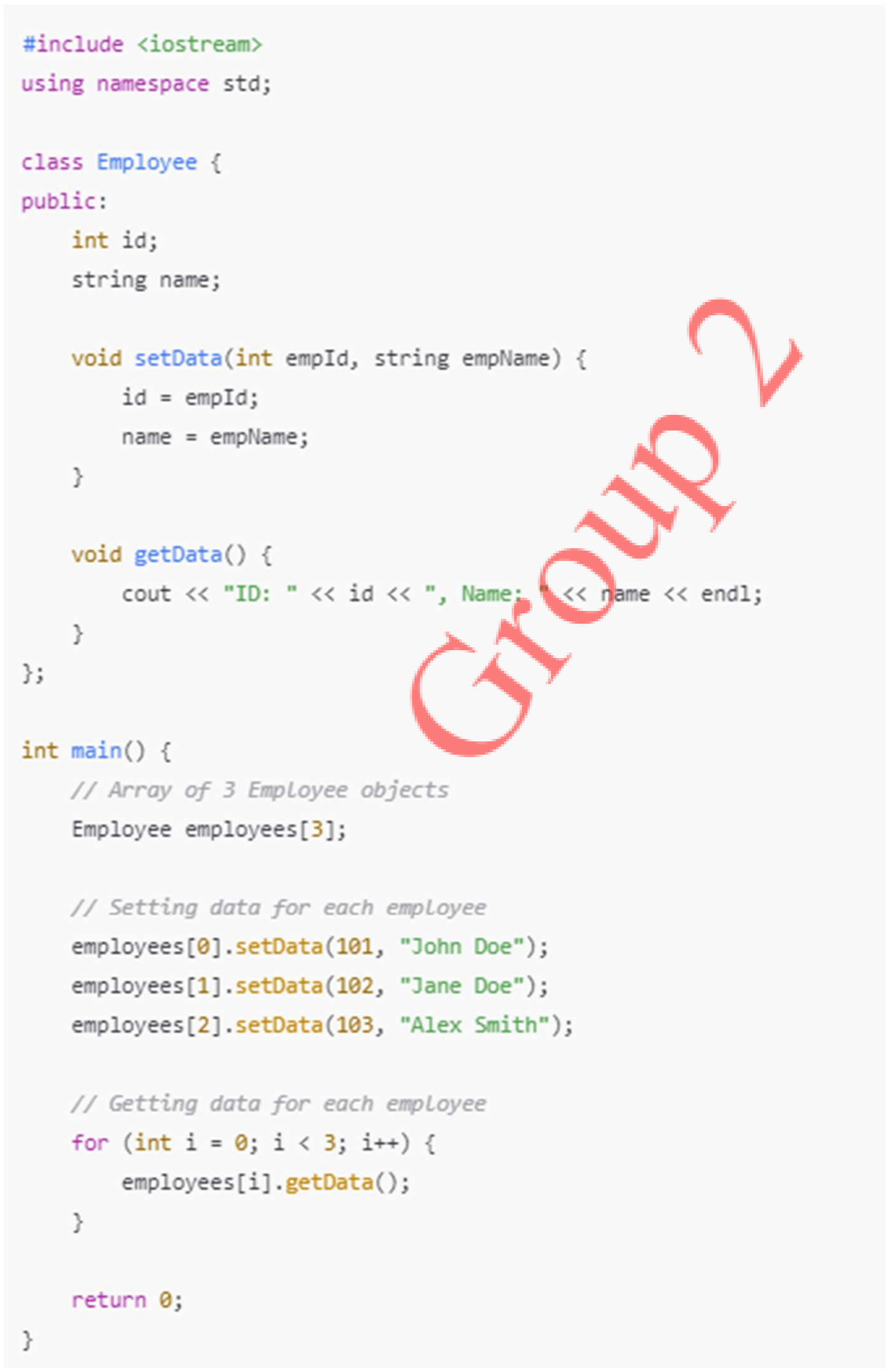


```cpp
#include <iostream>
using namespace std;

class Employee {
public:
    int id;
    string name;

    // Declaration inside the class
    void setData(int empId, string empName);
    void getData();
};

// Definition outside the class
void Employee::setData(int empId, string empName) {
    id = empId;
    name = empName;
}

void Employee::getData() {
    cout << "ID: " << id << ", Name: " << name << endl;
}

int main() {
    Employee emp2;
    emp2.setData(102, "Jane Doe");
    emp2.getData();
    return 0;
}
```

Array of Objects in C++

Definition:

An **array of objects** is an array where each element is an object of a class. This allows you to create multiple objects of the same class and access their members using array indexing.



```cpp
#include <iostream>
using namespace std;

class Employee {
public:
    int id;
    string name;

    void setData(int empId, string empName) {
        id = empId;
        name = empName;
    }

    void getData() {
        cout << "ID: " << id << ", Name: " << name << endl;
    }
};

int main() {
    // Array of 3 Employee objects
    Employee employees[3];

    // Setting data for each employee
    employees[0].setData(101, "John Doe");
    employees[1].setData(102, "Jane Doe");
    employees[2].setData(103, "Alex Smith");

    // Getting data for each employee
    for (int i = 0; i < 3; i++) {
        employees[i].getData();
    }

    return 0;
}
```

Attributes: The class has two attributes: id (integer) and name (string), representing the employee's ID and name.

setData() function: This method takes two parameters (empId and empName) to assign values to id and name.

getData() function: This method outputs the id and name of the employee to the console.

Encapsulation: The class encapsulates employee details and provides functions to set and display the data.

Here's a breakdown of the main() function:

1. **Array of Employee Objects**: An array employees is created to store 3 Employee objects.
2. **Setting Data**: The setData() function is called for each employee to set their id and name.
3. **Getting Data**: A loop iterates through each Employee object in the array, calling the getData() function to display their id and name.
4. **Output**: The details of each employee are printed to the console.

This code demonstrates how to manage and display data for multiple Employee objects using an array.