OOPS IN CPP

1. Inheritance:

Inheritance is a concept in Object-Oriented Programming (OOP) where one class (called a **child class**) can inherit properties and behaviors (i.e., attributes and methods) from another class (called a **parent class**). It helps in code reuse and establishing a relationship between classes.

- **Parent class**: This is the class from which other classes inherit.
- **Child class**: This is the class that inherits from another class.

Example:

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

// Parent class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

// Child class that inherits from Animal
class Dog : public Animal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};

int main() {
    Dog dog;  // Create a Dog object
    dog.eat();  // Dog can use the method from Animal class
    dog.bark();  // Dog has its own method
    return 0;
}
```

Explanation:

- `Animal` is the parent class, and `Dog` is the child class.
- The `Dog` class inherits the `eat()` method from the `Animal` class. So, when we create a `Dog` object, it can use the `eat()` method even though it's not defined inside the `Dog` class.

2. Polymorphism:

Polymorphism means "many forms." It allows you to use a common interface for different types of objects. In C++, polymorphism is achieved through **method overriding** (runtime polymorphism) and **method overloading** (compile-time polymorphism).

- **Runtime Polymorphism**: Achieved by overriding methods.
- **Compile-time Polymorphism**: Achieved by overloading methods or operators.

Example (Runtime Polymorphism):

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

// Base class
class Animal {
public:
    virtual void sound() {  // Virtual function allows overriding
        cout << "Animal makes a sound." << endl;
    }
};

// Derived class
class Dog : public Animal {
public:
    void sound() override {  // Override the base class method
        cout << "Dog barks." << endl;
    }
};

int main() {
    Animal* animal = new Dog();  // Base class pointer points to derived class
    animal->sound();  // The overridden method in Dog class is called
    delete animal;  // Clean up memory
    return 0;
}
```

Explanation:

- `sound()` is a method in both the `Animal` and `Dog` classes.
- Since the `sound()` method is **virtual** in the base class, when we call `sound()` on a base class pointer pointing to a `Dog` object, C++ will call the `Dog` class's version of the method. This is **runtime polymorphism**.

3. Abstraction:

Abstraction means hiding the complex implementation details and showing only the necessary features. It allows you to focus on what an object does, rather than how it does it.

In C++, abstraction is often achieved using **abstract classes**. An abstract class is a class that has at least one **pure virtual function** (a function declared with `= 0`), meaning that the class cannot be instantiated directly.

Example:

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

// Abstract class
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};

// Derived class
class Circle : public Shape {
public:
    void draw() override {  // Override the pure virtual function
        cout << "Drawing Circle" << endl;
    }
};

int main() {
    Shape* shape = new Circle();
    shape->draw();  // Call draw() of Circle
    delete shape;
    return 0;
}
```

Explanation:

- `Shape` is an abstract class with a pure virtual function `draw()`.
- We cannot create an object of `Shape` directly because it has a pure virtual function.
- `Circle` is a derived class that implements the `draw()` method. We can create a `Circle` object and call `draw()`.

4. Encapsulation:

Encapsulation is the concept of **bundling data** (attributes) and methods (functions) that operate on the data into a single unit, or class. It also involves **restricting direct access** to some of an object's components and exposing only the necessary functionality.

In C++, encapsulation is achieved by using **private** and **public** access specifiers:

- **Private** members are hidden and cannot be accessed outside the class directly.
- **Public** members can be accessed from outside the class.

Example:

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

class Account {
private:
    double balance;  // Private data member

public:
    // Public method to deposit money
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Public method to get balance
    double getBalance() {
        return balance;
    }
};

int main() {
    Account acc;
    acc.deposit(1000);  // Deposit money
    cout << "Balance: $" << acc.getBalance() << endl;  // Access balance
    return 0;
}
```

Explanation:

- The `balance` variable is private, so it cannot be accessed directly outside the `Account` class.
- The `deposit()` and `getBalance()` methods are public, allowing controlled access to the balance.

5. Function Overloading:

Function overloading allows you to define multiple functions with the same name but with different parameters (either different types or different numbers of parameters). The compiler will decide which function to call based on the arguments passed.

Example:

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

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    cout << "Sum of integers: " << calc.add(3, 4) << endl;  // Calls int version
    cout << "Sum of doubles: " << calc.add(2.5, 3.1) << endl;  // Calls double version
    return 0;
}
```

Explanation:

- The `add` function is overloaded. There are two versions: one that accepts two integers and one that accepts two doubles.
- The correct version of the `add()` function is chosen based on the type of the arguments.

6. Operator Overloading:

Operator overloading allows you to redefine the behavior of operators (like `+`, `-`, `=`, etc.) for user-defined types (classes).

Example:

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

class Complex {
private:
    int real, imag;

public:
    Complex(int r, int i) : real(r), imag(i) {}

    // Overload + operator
    Complex operator + (const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2, 3), c2(4, 5);
    Complex c3 = c1 + c2;  // Using overloaded + operator
    c3.display();  // Displays the sum of c1 and c2
    return 0;
}
```

Explanation:

- The `+` operator is overloaded in the `Complex` class. Now, you can use the `+` operator to add two `Complex` objects.
- The `operator+` function defines how the addition of two `Complex` numbers should be handled.

7. Method Overriding:

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in the parent class. The overridden method has the same signature as the one in the parent class.

Example:

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

class Animal {
public:
    virtual void sound() {
        cout << "Animal makes a sound." << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Dog barks." << endl;
    }
};

int main() {
    Animal* animal = new Dog();
    animal->sound();  // Calls the overridden method in Dog class
    delete animal;
    return 0;
}
```

Explanation:

- The `sound()` method in the `Animal` class is overridden in the `Dog` class.
- The `Dog` class provides its own version of the `sound()` method. This is **method overriding**.

Summary:

- **Inheritance** allows one class to inherit properties and methods from another.
- **Polymorphism** allows methods to behave differently based on the object calling them.
- **Abstraction** hides the details and shows only the essential features.
- **Encapsulation** bundles data and functions together and restricts access to some components.
- **Function Overloading** allows multiple functions with the same name but different parameters.
- **Operator Overloading** allows you to redefine the behavior of operators.
- **Method Overriding** allows a subclass to provide a specific implementation of a method already defined in the parent class.

These concepts are the foundation of OOP and will help you write cleaner, more maintainable code in C++.