Assignment:-3

```
// 1. Abstraction (Encapsulation)
#include <iostream>
using namespace std;
class BankAccount {
private:
    int accountNumber;
    float balance;
public:
    BankAccount(int accNo, float bal) : accountNumber(accNo), balance(bal) {}
    void deposit(float amount) { balance += amount; }
    void withdraw(float amount) {
        if (amount <= balance) balance -= amount;
    }
    void display() {
        cout << "Account: " << accountNumber << ", Balance: " << balance <<
endl;
    }
};

// 2. Data Hiding with Getters & Setters
class Student {
private:
    string name;
    int marks;
public:
    void setName(string n) { name = n; }
    void setMarks(int m) { marks = m; }
    string getName() { return name; }
    int getMarks() { return marks; }
};

// 3. Function Overloading
class MathOperations {
public:
    int multiply(int a, int b) { return a * b; }
    int multiply(int a, int b, int c) { return a * b * c; }
    float multiply(float a, int b) { return a * b; }
};

// 4. Operator Overloading (Unary)
class Counter {
private:
    int count;
public:
    Counter() : count(0) {}
    void display() { cout << "Count: " << count << endl; }
    Counter operator++() {
        ++count;
        return *this;
    }
};

// 5. Operator Overloading (Binary)
class Complex {
public:
    int real, imag;
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}
    Complex operator+(Complex const &obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }
    void display() { cout << real << "+" << imag << "i" << endl; }
};
```

```
// 6. Type Conversion
class Distance {
private:
    int feet, inches;
public:
    Distance(int f, int i) : feet(f), inches(i) {}
    operator int() { return feet * 12 + inches; }
};

// 7. Single Inheritance
class Animal {
public:
    virtual void makeSound() { cout << "Animal sound" << endl; }
};
class Dog : public Animal {
public:
    void makeSound() override { cout << "Bark" << endl; }
};

// 8. Multi-level Inheritance
class Person {
public:
    string name;
    int age;
};
class Employee : public Person {
public:
    float salary;
};
class Manager : public Employee {
public:
    string department;
};

// 9. Multiple Inheritance
class StudentMI {
public:
    string name;
    int rollNo;
};
class Sports {
public:
    int score;
};
class Result : public StudentMI, public Sports {
public:
    void display() {
        cout << "Name: " << name << ", RollNo: " << rollNo << ", Score: " <<
score << endl;
    }
};

// 10. Hierarchical Inheritance
class Vehicle {
public:
    void display() { cout << "This is a vehicle." << endl; }
};
class Car : public Vehicle {
public:
    void displayType() { cout << "Car" << endl; }
};
class Bike : public Vehicle {
public:
```

```cpp
    void displayType() { cout << "Bike" << endl; }
};

// 11. Hybrid (Virtual) Inheritance
class PersonH {
public:
    string name;
};
class Teacher : virtual public PersonH {
public:
    string subject;
};
class StudentH : virtual public PersonH {
public:
    int grade;
};
class TeachingAssistant : public Teacher, public StudentH {
public:
    void display() { cout << name << ", " << subject << ", Grade: " << grade <<
endl; }
};

// 12. Constructor and Destructor Order
class Parent {
public:
    Parent() { cout << "Parent Constructor\n"; }
    ~Parent() { cout << "Parent Destructor\n"; }
};
class Child : public Parent {
public:
    Child() { cout << "Child Constructor\n"; }
    ~Child() { cout << "Child Destructor\n"; }
};

// 13. Abstract Class
class Shape {
public:
    virtual float area() = 0;
};
class Circle : public Shape {
public:
    float radius;
    Circle(float r) : radius(r) {}
    float area() override { return 3.14 * radius * radius; }
};
class Rectangle : public Shape {
public:
    float length, width;
    Rectangle(float l, float w) : length(l), width(w) {}
    float area() override { return length * width; }
};

// 14. Method Overriding
class EmployeeBase {
public:
    virtual float calculateSalary() { return 1000; }
};
class ManagerOverride : public EmployeeBase {
public:
    float calculateSalary() override { return 3000; }
};

// 15. Virtual Functions
class AnimalBase {
```

```cpp
public:
    virtual void speak() { cout << "Animal speaks" << endl; }
};
class DogDerived : public AnimalBase {
public:
    void speak() override { cout << "Dog barks" << endl; }
};
class CatDerived : public AnimalBase {
public:
    void speak() override { cout << "Cat meows" << endl; }
};

// 16. Friend Function
class RectangleFF {
private:
    float length, breadth;
public:
    RectangleFF(float l, float b) : length(l), breadth(b) {}
    friend float calculateArea(RectangleFF);
};
float calculateArea(RectangleFF r) { return r.length * r.breadth; }

// 17. Static Members
class CounterStatic {
public:
    static int count;
    CounterStatic() { count++; }
    static void showCount() { cout << "Objects created: " << count << endl; }
};
int CounterStatic::count = 0;

// 18. Copy Constructor
class Book {
public:
    string title;
    Book(string t) : title(t) {}
    Book(const Book &b) { title = b.title; }
};

// 19. Exception Handling
class Division {
public:
    void divide(int a, int b) {
        try {
            if (b == 0) throw "Division by zero!";
            cout << "Result: " << a / b << endl;
        } catch (const char* msg) {
            cout << msg << endl;
        }
    }
};

// 20. Smart Pointer (unique_ptr)
#include <memory>
void smartPointerDemo() {
    unique_ptr<int> ptr(new int);
    *ptr = 10;
    cout << "Smart pointer value: " << *ptr << endl;
}
```