Create a class Person with private attributes name, age, and public methods to set and get the values.

```
#include <iostream>

using namespace std;

class Person {

private:

   string name;

   int age;

public:

   void setValues(string n, int a) {

      name = n;

      age = a;

   }

   void getValues() {

      cout << "Name: " << name << ", Age: " << age << endl;

   }

};

int main() {

   Person p;

   p.setValues("Sakshi", 19);

   p.getValues();

   return 0;

}
```

Implement a class Student that inherits from Person and adds a private attribute studentID with appropriate methods.

```
#include <iostream>
```

```
using namespace std;
class Person {
private:
    string name;
    int age;
public:
    void setValues(string n, int a) {
        name = n;
        age = a;
    }
    void getValues() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

class Student : public Person {
private:
    int studentID;
public:
    void setStudentID(int id) {
        studentID = id;
    }
    void getStudentDetails() {
        getValues();
        cout << "Student ID: " << studentID << endl;
```

```
    }
};
int main() {
    Student s;
    s.setValues("Shubhangi", 19);
    s.setStudentID(101);
    s.getStudentDetails();
    return 0;
}
```

Design a class Car with attributes for make, model, and year. Include methods to display car details.

```
#include <iostream>
using namespace std;
class Car {
private:
    string make, model;
    int year;
public:
    void setCar(string m, string mod, int y) {
        make = m;
        model = mod;
        year = y;
    }
    void displayCar() {
        cout << "Make: " << make << ", Model: " << model << ", Year: " << year << endl;
    }
```

```
};
int main() {
    Car c;
    c.setCar("Toyota", "Camry", 2022);
    c.displayCar();
    return 0;
}
```

Write a program that creates an array of Car objects and displays their details.

```
#include <iostream>
using namespace std;
class Car {
private:
    string make, model;
    int year;
public:
    void setCar(string m, string mod, int y) {
        make = m;
        model = mod;
        year = y;
    }
    void displayCar() {
        cout << "Make: " << make << ", Model: " << model << ", Year: " << year << endl;
    }
};
int main() {
```

```
    Car cars[3];
    cars[0].setCar("Toyota", "Camry", 2022);
    cars[1].setCar("Honda", "Civic", 2023);
    cars[2].setCar("Ford", "Mustang", 2021);
    for (int i = 0; i < 3; i++) {
        cars[i].displayCar();
    }
    return 0;
}
```

Implement a class BankAccount with private attributes for account number, balance, and public methods for depositing and withdrawing money.

```
#include <iostream>
using namespace std;
class BankAccount {
private:
    int accountNumber;
    double balance;

public:
    void setAccount(int accNum, double bal) {
        accountNumber = accNum;
        balance = bal;
    }
    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << ", New Balance: " << balance << endl;
```

```
    }
    void withdraw(double amount) {
        if (amount > balance) {
            cout << "Insufficient balance!" << endl;
        } else {
            balance -= amount;
            cout << "Withdrawn: " << amount << ", Remaining Balance: " << balance << endl;
        }
    }
    void display() {
        cout << "Account Number: " << accountNumber << ", Balance: " << balance << endl;
    }
};
int main() {
    BankAccount account;
    account.setAccount(123456, 5000);
    account.display();
    account.deposit(2000);
    account.withdraw(1000);
    account.withdraw(7000);
    return 0;
}
```

Create a class Rectangle with private attributes for length and width, and public methods to calculate area and perimeter.

```
class Rectangle {
private:
```

```
    double length, width;

public:
    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    double area() {
        return length * width;
    }

    double perimeter() {
        return 2 * (length + width);
    }
};
```

Write a class Employee with private attributes name, position, and salary, and public methods to display employee details.

```
class Employee {
private:
    string name, position;
    double salary;
public:
    Employee(string n, string p, double s) {
        name = n;
        position = p;
```

```
        salary = s;
    }
    void display() {
        cout << "Name: " << name << ", Position: " << position << ", Salary: " << salary << endl;
    }
};
```

Create a class Counter with a static data member to count the number of objects created.

```
class Counter {
public:
    static int count;
    Counter() {
        count++;
    }
};
int Counter::count = 0;
```

Implement a class Math with static function members for basic arithmetic operations.

```
class Math {
public:
    static int add(int a, int b) { return a + b; }
    static int subtract(int a, int b) { return a - b; }
    static int multiply(int a, int b) { return a * b; }
    static double divide(int a, int b) { return (double)a / b; }
};
```

Write a class Student with a static data member to keep track of the total number of students enrolled.

```
class Student {
```

```
public:
    static int totalStudents;
    Student() {
        totalStudents++;
    }
};
int Student::totalStudents = 0;
```

Implement a class Book with a parameterized constructor to initialize book details.

```
class Book {
public:
    string title, author;
    int pages;
    Book(string t, string a, int p) {
        title = t;
        author = a;
        pages = p;
    }
};
```

Create a class Point with a default constructor, parameterized constructor, and copy constructor.

```
class Point {
public:
    int x, y;
    Point() { x = y = 0; }
    Point(int a, int b) { x = a; y = b; }
    Point(const Point &p) { x = p.x; y = p.y; }
```

```
};
```

Write a class Matrix with a parameterized constructor to initialize a 2D array.

```
class Matrix {
private:
    int arr[2][2];

public:
    Matrix(int a, int b, int c, int d) {
        arr[0][0] = a; arr[0][1] = b;
        arr[1][0] = c; arr[1][1] = d;
    }

    void display() {
        cout << arr[0][0] << " " << arr[0][1] << endl;
        cout << arr[1][0] << " " << arr[1][1] << endl;
    }
};
```

Implement a class FileHandler with a destructor that closes an open file.

```
#include <fstream>
class FileHandler {
    fstream file;

public:
    FileHandler(string filename) {
        file.open(filename, ios::out);
```

```
    }

    ~FileHandler() {
        file.close();
        cout << "File closed.\n";
    }
};
```

Create a class DynamicArray with a destructor that deallocates dynamically allocated memory.

```
class DynamicArray {
    int *arr;
public:
    DynamicArray(int size) {
        arr = new int[size];
    }

    ~DynamicArray() {
        delete[] arr;
        cout << "Memory deallocated.\n";
    }
};
```

Write a class Logger with a destructor that logs messages when the object is destroyed.

```
class Logger {
public:
    ~Logger() {
        cout << "Logger: Object destroyed.\n";
```

```
    }
};
```

Overload the "+" operator for a class Complex to add two complex numbers.

```
class Complex {
public:
    int real, imag;

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

    Complex operator+(const Complex &c) {
        return Complex(real + c.real, imag + c.imag);
    }
};
```

Implement the "<<" and ">>" operators for a class Fraction to input and output fraction values.

```
class Fraction {
    int num, den;

public:
    friend istream &operator>>(istream &in, Fraction &f) {
        in >> f.num >> f.den;
        return in;
    }

    friend ostream &operator<<(ostream &out, Fraction &f) {
        out << f.num << "/" << f.den;
```

```
        return out;
    }
};
```

Overload the "==" operator for a class Date to compare two dates.

```
class Date {
    int day, month, year;

public:
    Date(int d, int m, int y) : day(d), month(m), year(y) {}

    bool operator==(const Date &d) {
        return day == d.day && month == d.month && year == d.year;
    }
};
```

Write a class Vector and overload the "[]" operator to access elements of the vector.

```
class Vector {
    int arr[10];

public:
    int &operator[](int index) {
        return arr[index];
    }
};
```

Implement a class Box with a friend function to calculate the volume of two boxes.

```
class Box {
```

```
    int length, width, height;

public:
    Box(int l, int w, int h) : length(l), width(w), height(h) {}

    friend int volume(Box b);
};

int volume(Box b) {
    return b.length * b.width * b.height;
}
```

Create a class Circle with a friend function to calculate the area.

```
class Circle {
    float radius;
public:
    Circle(float r) : radius(r) {}
    friend float area(Circle c);
};
float area(Circle c) {
    return 3.14 * c.radius * c.radius;
}
```

Write a class Distance with a friend function to add two distances.

```
class Distance {
    int meters;
public:
```

```
    Distance(int m) : meters(m) {}
    friend Distance add(Distance a, Distance b);
};
Distance add(Distance a, Distance b) {
    return Distance(a.meters + b.meters);
}
```

Implement a class Shape with derived classes Circle, Rectangle, and Triangle.

```
class Shape {
public:
    void display() {
        cout << "I am a shape\n";
    }
};
class Circle : public Shape {};
class Rectangle : public Shape {};
class Triangle : public Shape {};
```

Create a class Animal with derived classes Dog, Cat, and Bird.

```
class Animal {
public:
    void speak() {
        cout << "Animal sound\n";
    }
};
class Dog : public Animal {};
class Cat : public Animal {};
```

```
class Bird : public Animal {};
```

Write a class Vehicle with derived classes Car and Bike.

```
class Vehicle {
public:
    void start() {
        cout << "Vehicle starting\n";
    }
};
class Car : public Vehicle {};
class Bike : public Vehicle {};
```

Implement single inheritance with a base class Person and derived class Employee.

```
class Person {
public:
    string name;
};
class Employee : public Person {
public:
    int salary;
};
```

Create a class Parent and implement multiple inheritance with derived classes Child1 and Child2.

```
class Parent1 {
public:
    void show1() { cout << "Parent1\n"; }
};
```

```
class Parent2 {
public:
    void show2() { cout << "Parent2\n"; }
};

class Child : public Parent1, public Parent2 {};
```

Write a class Base and implement hierarchical inheritance with derived classes Derived1, Derived2, and Derived3.

```
class Base {
public:
    void show() { cout << "Base class\n"; }
};
class Derived1 : public Base {};
class Derived2 : public Base {};
class Derived3 : public Base {};
```

Implement multilevel inheritance with classes Base, Intermediate, and Derived.

```
class Base {
public:
    void show() { cout << "Base\n"; }
};
class Intermediate : public Base {};
class Derived : public Intermediate {};
```

Create a class Base and implement hybrid inheritance with derived classes Derived1, Derived2, and Derived3. #include <iostream>

```
using namespace std;
```

```
class Base {
public:
    void showBase() { cout << "Base class\n"; }
};

class Derived1 : virtual public Base {
public:
    void showD1() { cout << "Derived1\n"; }
};

class Derived2 : virtual public Base {
public:
    void showD2() { cout << "Derived2\n"; }
};

class Derived3 : public Derived1, public Derived2 {
public:
    void showD3() { cout << "Derived3\n"; }
};
```

Implement a class Library with private, protected, and public members and demonstrate their accessibility.

```
class Library {
private:
    int books;
protected:
    string name;
```

```
public:
    void setData(int b, string n) { books = b; name = n; }
    void showData() { cout << "Books: " << books << ", Name: " << name << endl; }
};
```

Create a class Account with private data members and public methods to access and modify them.

```
class Account {
private:
    int balance;
public:
    void setBalance(int b) { balance = b; }
    int getBalance() { return balance; }
};
```

Implement function overriding with a base class Shape and derived class Circle.

```
class Shape {
public:
    virtual void draw() { cout << "Drawing Shape\n"; }
};

class Circle : public Shape {
public:
    void draw() override { cout << "Drawing Circle\n"; }
};
```

Create a base class Employee and derived class Manager with overridden methods.

```
class Employee {
public:
```

```
    virtual void work() { cout << "Employee working\n"; }
};

class Manager : public Employee {
public:
    void work() override { cout << "Manager managing\n"; }
};
```

Implement a virtual base class Entity with derived classes Person and Organization.

```
class Entity {
public:
    void identity() { cout << "Entity\n"; }
};

class Person : virtual public Entity {};
class Organization : virtual public Entity {};
class Hybrid : public Person, public Organization {};
```

Write a class Animal and implement a virtual base class to avoid the diamond problem in inheritance.

```
class Animal {
public:
    void speak() { cout << "Animal speaking\n"; }
};

class Mammal : virtual public Animal {};
class Bird : virtual public Animal {};
class Bat : public Mammal, public Bird {};
```

Implement a class Polynomial with member functions to add and multiply polynomials.

```
class Polynomial {
   int coeff[3]; // ax^2 + bx + c
public:
   void set(int a, int b, int c) {
      coeff[0] = a; coeff[1] = b; coeff[2] = c;
   }
   void display() {
      cout << coeff[0] << "x^2 + " << coeff[1] << "x + " << coeff[2] << endl;
   }
   Polynomial add(Polynomial p) {
      Polynomial res;
      res.set(coeff[0] + p.coeff[0], coeff[1] + p.coeff[1], coeff[2] + p.coeff[2]);
      return res;
   }
   Polynomial multiply(Polynomial p) {
      Polynomial res;
      res.set(coeff[0] * p.coeff[0], coeff[1] * p.coeff[1], coeff[2] * p.coeff[2]);
      return res;
   }
};
```

Create a class SparseMatrix with member functions for matrix addition and multiplication.

```
class SparseMatrix {
   int mat[2][2];
public:
```

```
    void set(int a, int b, int c, int d) {
        mat[0][0] = a; mat[0][1] = b;
        mat[1][0] = c; mat[1][1] = d;
    }
    void display() {
        cout << mat[0][0] << " " << mat[0][1] << endl;
        cout << mat[1][0] << " " << mat[1][1] << endl;
    }
};
```

Write a class Time with member functions to add, subtract, and compare time values.

```
class Time {
    int h, m;
public:
    Time(int hour = 0, int min = 0) : h(hour), m(min) {}
    void add(Time t) {
        h += t.h; m += t.m;
        h += m / 60; m %= 60;
    }
    void subtract(Time t) {
        int t1 = h*60 + m;
        int t2 = t.h*60 + t.m;
        int diff = t1 - t2;
        h = diff / 60; m = diff % 60;
    }
    bool isEqual(Time t) {
```

```
        return h == t.h && m == t.m;
    }
    void display() {
        cout << h << " hr " << m << " min\n";
    }
};
```

Implement a class BigNumber to handle arithmetic operations on large numbers.

```
class BigNumber {
    string num;
public:
    BigNumber(string n) : num(n) {}
    void display() { cout << num << endl; }
    BigNumber add(BigNumber b) {
        string a = num, c = b.num, res = "";
        int carry = 0;
        int i = a.length() - 1, j = c.length() - 1;
        while (i >= 0 || j >= 0 || carry) {
            int sum = carry;
            if (i >= 0) sum += a[i--] - '0';
            if (j >= 0) sum += c[j--] - '0';
            carry = sum / 10;
            res = char(sum % 10 + '0') + res;
        }
        return BigNumber(res);
    }
```

};

Create a class FileCompressor with member functions to compress and decompress files.

```
class FileCompressor {
public:
    void compress() { cout << "Compressing file...\n"; }
    void decompress() { cout << "Decompressing file...\n"; }
};
```

Write a class Network with member functions to simulate network packet transmission.

```
class Network {
public:
    void sendPacket() { cout << "Sending packet...\n"; }
    void receivePacket() { cout << "Receiving packet...\n"; }
};
```

Implement a class Cache with member functions to store and retrieve cached data.

```
class Cache {
    string data;
public:
    void store(string d) { data = d; }
    string retrieve() { return data; }
};
```

Create a class Game with member functions to simulate a simple game with player actions and scoring.

```
class Game {
    int score;
public:
    Game() : score(0) {}
```

```
    void action(string act) {
        if (act == "jump") score += 10;
        else if (act == "run") score += 5;
    }
    void showScore() { cout << "Score: " << score << endl; }
};
```