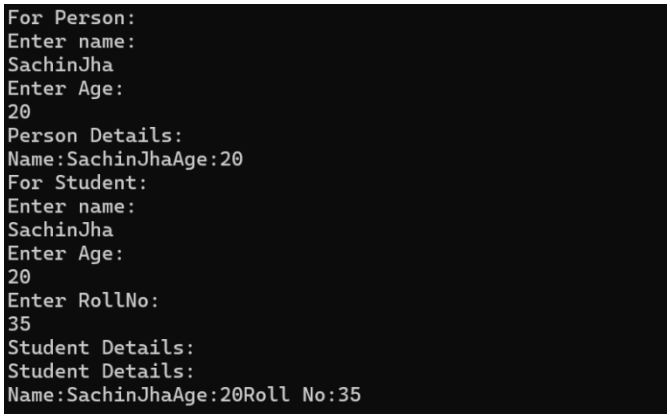
LAB QUESTIONS:

1. Write a C++ program to create a base class Person with attributes name and age. Derive a class Student that adds rollNo. Use constructors to initialize all attributes. Create objects of both classes and display their details to show how Student inherits Person members**.**

```
#include<iostream>
using namespace std;
class Person{
protected:
   string name;
   int age;
public:
   Person(string n, int a){
   name=n;
   age=a;
   }
   void displayInfo(){
   cout<<"Name:"<<name<<"Age:"<<age;
   }};
class Student: public Person{
private:
   int roll;
public:
   Student(string n, int a, int r):Person(n,a),roll(r){
   }
   void displayInfo(){
   cout<<"Student Details:"<<endl;
   Person::displayInfo();
   cout<<"Roll No:"<<roll<<endl;
   }};
int main(){
```

```
string name;
int age; int roll;
cout<<"For Person:"<<endl;
cout<<"Enter name:"<<endl;
cin>>name;
cout<<"Enter Age:"<<endl;
cin>>age;
Person p(name,age);
cout<<"Person Details:"<<endl;
p.displayInfo();
cout<<endl;
cout<<"For Student:"<<endl;
cout<<"Enter name:"<<endl;
cin>>name;
cout<<"Enter Age:"<<endl;
cin>>age;
cout<<"Enter RollNo:"<<endl;
cin>>roll;
Student s(name,age,roll);
cout<<"Student Details:"<<endl;
s.displayInfo();
return 0;
}
```



2. Implement a C++ program with a base class Account having a protected attribute balance. Derive a class SavingsAccount that adds an attribute interestRate and a function addInterest() to modify balance. Use user input to initialize attributes and show how the protected balance is accessed in the derived class but not outside.

```
#include <iostream>

using namespace std;

class Account {

protected:

   double balance;

public:

   Account(double bal) {

      balance = bal;

   }

   void showBalance() {

      cout << "Current Balance: " << balance << endl;

   }};

class SavingsAccount : public Account {

private:

   double interestRate;

public:

   SavingsAccount(double bal, double rate) : Account(bal) {

      interestRate = rate;

   }

   void addInterest() {

      double interest = (balance * interestRate) / 100;

      balance += interest;

   }

   void display() {

      cout << "Balance after adding interest: " << balance << endl;

   }};

int main() {
```

```
    double bal, rate;
    cout << "Enter initial balance: ";
    cin >> bal;
    cout << "Enter interest rate (%): ";
    cin >> rate;
    SavingsAccount sAcc(bal, rate);
    sAcc.addInterest();
    sAcc.display();
    return 0;
}
```

```
Enter initial balance: 35000
Enter interest rate (%): 4
Balance after adding interest: 36400
```

3. Write a C++ program with a base class Shape having a function draw(). Declare a derived class Circle with an attribute radius initialized via user input. Create a Circle object and call draw() to display a message including radius, demonstrating proper derived class declaration.

```
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape." << endl;
    }
};
class Circle : public Shape {
private:
    float radius;
public:
    Circle() {
        cout << "Enter radius: ";
        cin >> radius;
    }
```

```
    void draw() override {
        cout << "Drawing a circle with radius: " << radius << endl;
    }
};
int main() {
    Circle c;
    c.draw();
    return 0;
}
```

```
Enter radius: 4
Drawing a circle with radius: 4
```

4. Create a C++ program with a base class Vehicle having a function move(). Derive a class Car that overrides move() to indicate driving. Use a base class pointer to call move() on a Car object initialized with user input for attributes like brand. Show that Car is a Vehicle.

```
#include <iostream>
using namespace std;
class Vehicle {
public:
    virtual void move() {
        cout << "Vehicle is moving." << endl;
    }
};
class Car : public Vehicle {
private:
    string brand;
public:
    Car() {
        cout << "Enter car brand: ";
        cin >> brand;
    }
```

```
    void move() override {
        cout << brand << " car is driving." << endl;
    }
};
int main() {
    Vehicle* v;
    Car c;
    v = &c;
    v->move();
    return 0;
}
```

```
Enter car brand: BMW
BMW car is driving.
```

5. Implement a C++ program with a class Engine having an attribute horsepower. Create a class Car that contains an Engine object (composition) and an attribute model. Initialize all attributes with user input and display details to show that Car has an Engine.

```
#include <iostream>
using namespace std;
class Engine {
private:
    int horsepower;
public:
    Engine() {
        cout << "Enter engine horsepower: ";
        cin >> horsepower;
    }
    void display() {
        cout << "Engine horsepower: " << horsepower << endl;
    }
};
```

```
class Car {
private:
    string model;
    Engine engine;
public:
    Car() : engine() {
        cout << "Enter car model: ";
        cin >> model;
    }
    void display() {
        cout << "Car model: " << model << endl;
        engine.display();
    }
};
int main() {
    Car c;
    c.display();
    return 0;
}
```

```
Enter engine horsepower: 746
Enter car model: B7A
Car model: B7A
Engine horsepower: 746
```

6. Write a C++ program with a base class Base having public, protected, and private attributes (e.g., pubVar, protVar, privVar). Derive three classes using public, protected, and private inheritance, respectively. Demonstrate with user-initialized objects how each inheritance type affects access to base class members.

```
#include <iostream>
#include <string>
using namespace std;
class Base {
```

```
public:
int pubVar;
protected:
int protVar;
private:
int privVar;
public:
Base(int pub, int prot, int priv) : pubVar(pub), protVar(prot), privVar(priv) {}
void display() const {
cout << "Base: Public Var = " << pubVar << ", Protected Var = " << protVar << ", Private Var = " <<privVar << endl;
}
};
class PublicDerived : public Base {
public:
PublicDerived(int pub, int prot, int priv) : Base(pub, prot, priv) {}
void display() const {
cout << "PublicDerived: Public Var = " << pubVar << ", Protected Var = " << protVar << endl;
}
};
class ProtectedDerived : protected Base {
public:
ProtectedDerived(int pub, int prot, int priv) : Base(pub, prot, priv) {}
void display() const {
cout << "ProtectedDerived: Public Var = " << pubVar << ", Protected Var = " << protVar << endl;
}
};
class PrivateDerived : private Base {
public:
PrivateDerived(int pub, int prot, int priv) : Base(pub, prot, priv) {}
```

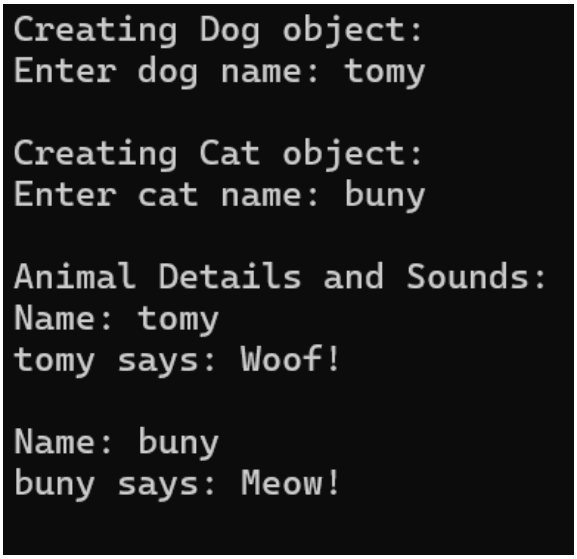
```
void display() const {
cout << "PrivateDerived: Public Var = " << pubVar << ", Protected Var = " << protVar << endl;
}
};
int main() {
int pub, prot, priv;
cout << "Creating Base object:" << endl;
cout << "Enter public variable: ";
cin >> pub;
cout << "Enter protected variable: ";
cin >> prot;
cout << "Enter private variable: ";
cin >> priv;
Base base(pub, prot, priv);
cout << "\nBase Object Details:" << endl;
base.display();
cout << "Accessing pubVar directly: " << base.pubVar << endl;
cout << endl;
cout << "Creating PublicDerived object:" << endl;
cout << "Enter public variable: ";
cin >> pub;
cout << "Enter protected variable: ";
cin >> prot;
cout << "Enter private variable: ";
cin >> priv;
PublicDerived pubDerived(pub, prot, priv);
cout << "\nPublicDerived Object Details:" << endl;
pubDerived.display();
cout << "Accessing pubVar directly: " << pubDerived.pubVar << endl;
```

```
cout << endl;
cout << "Creating ProtectedDerived object:" << endl;
cout << "Enter public variable: ";
cin >> pub;
cout << "Enter protected variable: ";
cin >> prot;
cout << "Enter private variable: ";
cin >> priv;
ProtectedDerived protDerived(pub, prot, priv);
cout << "\nProtectedDerived Object Details:" << endl;
protDerived.display();
cout << endl;
cout << "Creating PrivateDerived object:" << endl;
cout << "Enter public variable: ";
cin >> pub;
cout << "Enter protected variable: ";
cin >> prot;
cout << "Enter private variable: ";
cin >> priv;
PrivateDerived privDerived(pub, prot, priv);
cout << "\nPrivateDerived Object Details:" << endl;
privDerived.display();
return 0;
}
```

7. Create a C++ program with a base class Animal having a virtual function sound(). Derive classes Dog and Cat that override sound() to print specific sounds. Use a base class pointer array to call sound() on Dog and Cat objects created with user input, showing runtime polymorphism.

```
#include <iostream>
#include <string>
using namespace std;
class Animal {
protected:
string name;
public:
Animal(string n) : name(n) {}
virtual void sound() const {
cout << name << " makes a generic animal sound." << endl;
}
void display() const {
cout << "Name: " << name << endl;
}
};
class Dog : public Animal {
public:
Dog(string n) : Animal(n) {}
void sound() const override {
cout << name << " says: Woof!" << endl;
}
};
class Cat : public Animal {
public:
Cat(string n) : Animal(n) {}
void sound() const override {
```

```
cout << name << " says: Meow!" << endl;
}
};
int main() {
string name;
Animal* animals[2];
cout << "Creating Dog object:" << endl;
cout << "Enter dog name: ";
getline(cin, name);
animals[0] = new Dog(name);
cout << "\nCreating Cat object:" << endl;
cout << "Enter cat name: ";
getline(cin, name);
animals[1] = new Cat(name);
cout << "\nAnimal Details and Sounds:" << endl;
for (int i = 0; i < 2; i++) {
animals[i]->display();
animals[i]->sound();
cout << endl;
}
for (int i = 0; i < 2; i++) {
delete animals[i];
}
return 0;
}
```



8. Write a C++ program with two base classes Battery and Screen, each with a function showStatus(). Derive a class Smartphone that inherits from both. Resolve ambiguity when calling showStatus() using the scope resolution operator. Initialize attributes with user input and display details.

```
#include <iostream>

using namespace std;

class Battery {

protected:

   int capacity;

public:

   void showStatus() {

      cout << "Battery capacity: " << capacity << " mAh\n";

   }

};

class Screen {

protected:

   float size;

public:

   void showStatus() {

      cout << "Screen size: " << size << " inches\n";

   }

};

class Smartphone : public Battery, public Screen {

   string brand;

public:

   void input() {

      cout << "Enter brand: ";

      cin >> brand;

      cout << "Enter battery capacity (mAh): ";

      cin >> capacity;

      cout << "Enter screen size (inches): ";
```

```
        cin >> size;
    }
    void display() {
        cout << "Brand: " << brand << "\n";
        Battery::showStatus();
        Screen::showStatus();
    }
};
int main() {
    Smartphone phone;
    phone.input();
    phone.display();
    return 0;
}
```

```
Enter brand: Samsung
Enter battery capacity (mAh): 50000
Enter screen size (inches): 7
Brand: Samsung
Battery capacity: 50000 mAh
Screen size: 7 inches
```

9. Implement a C++ program with a base class Person having a parameterized constructor for name and age. Derive a class Employee with an additional attribute employeeID. Use user input to initialize all attributes and show the order of constructor invocation when creating an Employee object.

```
#include <iostream>
#include <string>
using namespace std;
class Person {
protected:
string name;
int age;
public:
Person(string n, int a) : name(n), age(a) {
cout << "Person constructor called: Name = " << name << ", Age = " << age << endl;
}
void display() const {
cout << "Name: " << name << ", Age: " << age << endl;
}
};
class Employee : public Person {
private:
string employeeID;
public:
Employee(string n, int a, string id) : Person(n, a), employeeID(id) {
cout << "Employee constructor called: EmployeeID = " << employeeID << endl;
}
void display() const {
cout << "Employee Details:" << endl;
Person::display();
cout << "Employee ID: " << employeeID << endl;
}
```

```
};
int main() {
string name, employeeID;
int age;
cout << "Creating Employee object:" << endl;
cout << "Enter name: ";
getline(cin, name);
cout << "Enter age: ";
cin >> age;
cin.ignore();
cout << "Enter employee ID: ";
getline(cin, employeeID);
cout << "\nConstructor Invocation Order:" << endl;
Employee employee(name, age, employeeID);
cout << "\n";
employee.display();
return 0;
}
```

```
Creating Employee object:
Enter name: Sachin jha
Enter age: 20
Enter employee ID: 0045

Constructor Invocation Order:
Person constructor called: Name = Sachin jha, Age = 20
Employee constructor called: EmployeeID = 0045

Employee Details:
Name: Sachin jha, Age: 20
Employee ID: 0045
```

10.Write a C++ program with a base class Shape and a derived class Rectangle, both with destructors that print messages. Make the base class destructor virtual. Create a Rectangle object through a base class pointer using user input for attributes, and delete it to show proper destructor invocation. Compare with a non-virtual destructor case.

```
#include <iostream>

using namespace std;

class Shape {

public:

    virtual ~Shape() {

        cout << "Shape destructor called (virtual)" << endl;

    }

};

class Rectangle : public Shape {

    int width, height;

public:

    Rectangle(int w, int h) : width(w), height(h) { }

    ~Rectangle() {

        cout << "Rectangle destructor called" << endl;

    }

    void display() {

        cout << "Rectangle: width = " << width << ", height = " << height << endl;

    }

};


int main() {

    int w, h;

    cin >> w >> h;

    Shape* shapePtr = new Rectangle(w, h);

    Rectangle* rectPtr = dynamic_cast<Rectangle*>(shapePtr);

    if (rectPtr) {

        rectPtr->display();
```

```
    }
    delete shapePtr;
    return 0;
}
```

```
5
5
Rectangle: width = 5, height = 5
Rectangle destructor called
Shape destructor called (virtual)
```