Q1) 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.

Code:

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

class Person {
protected:
    string name;
    int age;

public:
    Person(string n, int a) {
        name = n;
        age = a;
    }

    void displayPerson() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

class Student : public Person {
private:
    int rollNo;

public:
    Student(string n, int a, int r) : Person(n, a) {
        rollNo = r;
    }

    void displayStudent() {
        displayPerson();
        cout << "Roll Number: " << rollNo << endl;
    }
};

int main() {
    string name;
    int age, rollNo;

    cout << "Enter details for a Person:" << endl;
    cout << "Name: ";
    getline(cin, name);
    cout << "Age: ";
    cin >> age;
    cin.ignore();

    Person person(name, age);
    cout << "\nPerson Details:" << endl;
    person.displayPerson();

    cout << "\nEnter details for a Student:" << endl;
    cout << "Name: ";
    getline(cin, name);
    cout << "Age: ";
    cin >> age;
    cout << "Roll Number: ";
    cin >> rollNo;

    Student student(name, age, rollNo);
    cout << "\nStudent Details:" << endl;
    student.displayStudent();

    return 0;
}
```

Code:

```
Enter details for a Person:
Name: alish
Age: 16

Person Details:
Name: alish
Age: 16

Enter details for a Student:
Name: aditya
Age: 19
Roll Number: 4

Student Details:
Name: aditya
Age: 19
Roll Number: 4
```

Q2) 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.

Code:

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

class Account {
protected:
    double balance;
public:
    Account(double initialBalance) {
        balance = initialBalance;
    }
    void displayBalance() const {
        cout << "Current Balance: Rs. " << balance << endl;
    }
};

class SavingsAccount : public Account {
private:
    double interestRate;
public:
    SavingsAccount(double initialBalance, double rate)
        : Account(initialBalance), interestRate(rate) {}
    void addInterest() {
        double interest = balance * interestRate / 100;
        balance += interest;
        cout << "Interest of Rs. " << interest << " added." << endl;
    }
    void showUpdatedBalance() {
        cout << "Updated Balance after interest: Rs. " << balance << endl;
    }
};

int main() {
    double balanceInput, rateInput;
    cout << "Enter initial balance: Rs. ";
    cin >> balanceInput;
    cout << "Enter interest rate (%): ";
    cin >> rateInput;
    SavingsAccount myAccount(balanceInput, rateInput);
    myAccount.addInterest();
    myAccount.showUpdatedBalance();
    return 0;
}
```

Output:

```
Enter initial balance: Rs. 12000
Enter interest rate (%): 12
Interest of Rs. 1440 added.
Updated Balance after interest: Rs. 13440
```

Q3) 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.

Code:

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

class Shape {
public:
    void draw() {
        cout << "Drawing a shape." << endl;
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) {
        radius = r;
    }

    void draw() {
        cout << "Drawing a circle with radius: " << radius << endl;
    }
};

int main() {
    double r;
    cout << "Enter radius of the circle: ";
    cin >> r;

    Circle c(r);
    c.draw();

    return 0;
}
```

Output:

```
Enter radius of the circle: 2
Drawing a circle with radius: 2
```

Q4) 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.

Code:

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

class Vehicle {
public:
    virtual void move() {
        cout << "The vehicle is moving." << endl;
    }
};

class Car : public Vehicle {
private:
    string brand;

public:
    Car(string b) {
        brand = b;
    }

    void move() override {
        cout << "The car (" << brand << ") is driving on the road." << endl;
    }
};

int main() {
    string userBrand;
    cout << "Enter the brand of the car: ";
    cin >> userBrand;

    Car myCar(userBrand);
    Vehicle* vehiclePtr = &myCar;

    vehiclePtr->move();

    return 0;
}
```

Output:

```
Enter the brand of the car: tesla
The car (tesla) is driving on the road.
```

Q5) 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.

Code:

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

class Engine {
private:
    int horsepower;

public:
    Engine(int hp) {
        horsepower = hp;
    }

    int getHorsepower() const {
        return horsepower;
    }
};

class Car {
private:
    string model;
    Engine engine;

public:
    Car(string m, int hp) : model(m), engine(hp) {}

    void displayDetails() const {
        cout << "Car Model: " << model << endl;
        cout << "Engine Horsepower: " << engine.getHorsepower() << " HP" << endl;
    }
};

int main() {
    string modelInput;
    int horsepowerInput;

    cout << "Enter car model: ";
    getline(cin, modelInput);

    cout << "Enter engine horsepower: ";
    cin >> horsepowerInput;

    Car myCar(modelInput, horsepowerInput);
    myCar.displayDetails();

    return 0;
}
```

Output:

```
Enter car model: toyota
Enter engine horsepower: 1000
Car Model: toyota
Engine Horsepower: 1000 HP
```

Q6) 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.

Code:

```
#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;
    }
};
```

```cpp
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;
}
```

Output:

```
Creating Base object:
Enter public variable: 4
Enter protected variable: 435
Enter private variable: 4

Base Object Details:
Base: Public Var = 4, Protected Var = 435, Private Var = 4
Accessing pubVar directly: 4

Creating PublicDerived object:
Enter public variable: 523
Enter protected variable: 54
Enter private variable: 45

PublicDerived Object Details:
PublicDerived: Public Var = 523, Protected Var = 54
Accessing pubVar directly: 523

Creating ProtectedDerived object:
Enter public variable: 423
Enter protected variable: 5435
Enter private variable: 2

ProtectedDerived Object Details:
ProtectedDerived: Public Var = 423, Protected Var = 5435

Creating PrivateDerived object:
Enter public variable: 4
Enter protected variable: 43
Enter private variable: 234

PrivateDerived Object Details:
PrivateDerived: Public Var = 4, Protected Var = 43
```

Q7) 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.

Code:

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

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

    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    void sound() const override {
        cout << "Dog says: Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void sound() const override {
        cout << "Cat says: Meow!" << endl;
    }
};

int main() {
    const int SIZE = 2;
    Animal* animals[SIZE];
    animals[0] = new Dog();
    animals[1] = new Cat();
    cout << "Animal sounds:" << endl;
    for (int i = 0; i < SIZE; ++i) {
        animals[i]->sound();
    }
    for (int i = 0; i < SIZE; ++i) {
        delete animals[i];
    }
    return 0;
}
```

Output:

```
Animal sounds:
Dog says: Woof!
Cat says: Meow!
```

Q8) 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.

Code:

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

class Battery {
protected:
    int batteryLevel;

public:
    Battery(int level) {
        batteryLevel = level;
    }

    void showStatus() const {
        cout << "Battery Level: " << batteryLevel << "%" << endl;
    }
};

class Screen {
protected:
    float size; // in inches

public:
    Screen(float s) {
        size = s;
    }

    void showStatus() const {
        cout << "Screen Size: " << size << " inches" << endl;
    }
};

class Smartphone : public Battery, public Screen {
private:
    string model;

public:
    Smartphone(string m, int level, float s)
        : Battery(level), Screen(s), model(m) {}

    void display() const {
        cout << "Smartphone Model: " << model << endl;
        Battery::showStatus();
        Screen::showStatus();
    }
};

int main() {
    string model;
    int level;
    float size;

    cout << "Enter smartphone model: ";
    getline(cin, model);

    cout << "Enter battery level (%): ";
    cin >> level;

    cout << "Enter screen size (in inches): ";
    cin >> size;

    Smartphone phone(model, level, size);

    cout << "\nSmartphone Details:\n";
    phone.display();

    return 0;
}
```

Output:

```
Enter smartphone model: S24
Enter battery level (%): 78
Enter screen size (in inches): 6.7

Smartphone Details:
Smartphone Model: S24
Battery Level: 78%
Screen Size: 6.7 inches
```

Q9) 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.

Code:

```cpp
#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." << endl;
    }

    void displayPerson() const {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

class Employee : public Person {
private:
    int employeeID;

public:
    Employee(string n, int a, int id) : Person(n, a), employeeID(id) {
        cout << "Employee constructor called." << endl;
    }

    void displayEmployee() const {
        displayPerson();
        cout << "Employee ID: " << employeeID << endl;
    }
};

int main() {
    string name;
    int age, id;

    cout << "Enter name: ";
    getline(cin, name);

    cout << "Enter age: ";
    cin >> age;

    cout << "Enter employee ID: ";
    cin >> id;

    cout << "\nCreating Employee object...\n" << endl;
    Employee emp(name, age, id);

    cout << "\nEmployee Details:\n";
    emp.displayEmployee();

    return 0;
}
```

Output:

```
Enter name: saswot
Enter age: 18
Enter employee ID: 166001

Creating Employee object...

Person constructor called.
Employee constructor called.

Employee Details:
Name: saswot
Age: 18
Employee ID: 166001
```

Q10) 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.

Code:

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

class Shape {
protected:
    double width;

public:
    Shape(double w) : width(w) {
        cout << "Shape constructor called: Width = " << width << endl;
    }

    virtual ~Shape() {
        cout << "Shape destructor called" << endl;
    }

    void display() const {
        cout << "Shape Width: " << width << endl;
    }
};

class Rectangle : public Shape {
private:
    double height;

public:
    Rectangle(double w, double h) : Shape(w), height(h) {
        cout << "Rectangle constructor called: Height = " << height << endl;
    }

    ~Rectangle() {
        cout << "Rectangle destructor called" << endl;
    }

    void display() const {
        cout << "Rectangle Details:" << endl;
        Shape::display();
        cout << "Height: " << height << endl;
    }
};

int main() {
    double width, height;

    cout << "Creating Rectangle object via base class pointer:" << endl;
    cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;

    Shape* shapePtr = new Rectangle(width, height);
    cout << "\n";
    shapePtr->display();

    cout << "\nDeleting Rectangle object via base class pointer:" << endl;
    delete shapePtr;

    return 0;
}
```

Output:

```
Creating Rectangle object via base class pointer:
Enter width: 7
Enter height: 3
Shape constructor called: Width = 7
Rectangle constructor called: Height = 3

Shape Width: 7

Deleting Rectangle object via base class pointer:
Rectangle destructor called
Shape destructor called
```

Q11) Create a C++ program with a base class A having an attribute value. Derive classes B and C from A, and derive class D from both B and C. Use virtual inheritance to avoid duplication of A's members. Initialize value with user input and display it from D to show ambiguity resolution.

Code:

```
#include <iostream>
using namespace std;
class A {
protected:
    int value;
public:
    A(int v) : value(v) {
        cout << "A constructor called: value = " << value << endl;
    }
    void showValue() const {
        cout << "Value from class A: " << value << endl;
    }
};
class B : virtual public A {
public:
    B(int v) : A(v) {
        cout << "B constructor called" << endl;
    }
};
class C : virtual public A {
public:
    C(int v) : A(v) {
        cout << "C constructor called" << endl;
    }
};
class D : public B, public C {
public:
    D(int v) : A(v), B(v), C(v) {
        cout << "D constructor called" << endl;
    }
    void display() const {
        showValue(); // No ambiguity due to virtual inheritance
    }
};
int main() {
    int val;
    cout << "Enter a value: ";
    cin >> val;
    D obj(val);
    cout << "\nAccessing value from class D:\n";
    obj.display();
    return 0;
}
```

Output:

```
Enter a value: 12
A constructor called: value = 12
B constructor called
C constructor called
D constructor called

Accessing value from class D:
Value from class A: 12
```

Discussion:

This lab focused on understanding how different types of type conversions work in C++, including conversions between basic types, from basic to user-defined types, and between user-defined types. We explored how constructors, conversion functions, and casting can be used to perform these conversions effectively. Additionally, we practiced inheritance concepts such as base and derived class relationships, protected access specifiers, and how derived classes inherit properties and behaviors from base classes. We also discussed the Is-a and Has-a relationships and how they apply in real program design. Understanding these principles helped us avoid ambiguity and better organize code for reusability and maintainability.

Conclusion:

From this lab, we gained practical insight into how type conversion enables flexible data handling in object-oriented programming. We also reinforced core inheritance concepts, including different types of inheritance, access control, and class relationships. These concepts form the foundation of reusable and structured C++ applications, enabling better code organization and extension through derived classes.