Lab 7(Understanding the Concept of Virtual Function, Virtual Base Class, and RTTI)

1. **Write a program to create a class `shape` with functions to find the area of the shapes and display the names of the shapes and other essential components of the class. Create derived classes `circle`, `rectangle`, and `trapezoid` each having overriding functions `area()` and `display()`. Write a suitable program to illustrate virtual functions and virtual destructors.**

```
#include <iostream>
#include <string>

class Shape {
protected:
    std::string name;

public:
    Shape(const std::string& shapeName) : name(shapeName) {}

    virtual double area() const = 0;

    virtual void display() const {
        std::cout << "Shape: " << name << std::endl;
    }

    virtual ~Shape() {
        std::cout << "Destroying shape: " << name << std::endl;
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(const std::string& shapeName, double circleRadius) : Shape(shapeName),
          radius(circleRadius) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void display() const override {
        std::cout << "Shape: " << name << std::endl;
        std::cout << "Type: Circle" << std::endl;
```

```
        std::cout << "Radius: " << radius << std::endl;
        std::cout << "Area: " << area() << std::endl;
    }

    ~Circle() override {
        std::cout << "Destroying Circle: " << name << std::endl;
    }
};

class Rectangle : public Shape {
private:
    double width;
    double height;

public:
    Rectangle(const std::string& shapeName, double rectWidth, double rectHeight) :
          Shape(shapeName), width(rectWidth), height(rectHeight) {}

    double area() const override {
        return width * height;
    }

    void display() const override {
        std::cout << "Shape: " << name << std::endl;
        std::cout << "Type: Rectangle" << std::endl;
        std::cout << "Width: " << width << std::endl;
        std::cout << "Height: " << height << std::endl;
        std::cout << "Area: " << area() << std::endl;
    }

    ~Rectangle() override {
        std::cout << "Destroying Rectangle: " << name << std::endl;
    }
};

class Trapezoid : public Shape {
private:
    double base1;
    double base2;
    double height;

public:
    Trapezoid(const std::string& shapeName, double trapBase1, double trapBase2,
          double trapHeight) : Shape(shapeName), base1(trapBase1),
          base2(trapBase2), height(trapHeight) {}
```

```
    double area() const override {
        return (base1 + base2) * height / 2.0;
    }

    void display() const override {
        std::cout << "Shape: " << name << std::endl;
        std::cout << "Type: Trapezoid" << std::endl;
        std::cout << "Base1: " << base1 << std::endl;
        std::cout << "Base2: " << base2 << std::endl;
        std::cout << "Height: " << height << std::endl;
        std::cout << "Area: " << area() << std::endl;
    }

    ~Trapezoid() override {
        std::cout << "Destroying Trapezoid: " << name << std::endl;
    }
};

int main() {
    Shape* shape1 = new Circle("Circle 1", 5.0);
    Shape* shape2 = new Rectangle("Rectangle 1", 3.0, 4.0);
    Shape* shape3 = new Trapezoid("Trapezoid 1", 2.0, 4.0, 3.0);

    shape1->display();
    std::cout << std::endl;
    shape2->display();
    std::cout << std::endl;
    shape3->display();
    std::cout << std::endl;

    delete shape1;
    delete shape2;
    delete shape3;

    return 0;
}
```

Output:

```
Shape: Circle 1
Type: Circle
Radius: 5
Area: 78.5397

Shape: Rectangle 1
```

```
Type: Rectangle
Width: 3
Height: 4
Area: 12

Shape: Trapezoid 1
Type: Trapezoid
Base1: 2
Base2: 4
Height: 3
Area: 9

Destroying Circle: Circle 1
Destroying shape: Circle 1
Destroying Rectangle: Rectangle 1
Destroying shape: Rectangle 1
Destroying Trapezoid: Trapezoid 1
Destroying shape: Trapezoid 1
```

2. **Create a class `Person` and two derived classes `Employee` and `Student`, inherited from class `Person`. Now create a class `Manager` which is derived from two base classes `Employee` and `Student`. Show the use of the virtual base class.**

```
#include <iostream>
#include <string>

class Person {
protected:
    std::string name;

public:
    Person(const std::string& n) : name(n) {}

    virtual void display() const {
        std::cout << "Person: " << name << std::endl;
    }
};

class Employee : virtual public Person {
protected:
    int employeeID;

public:
    Employee(const std::string& n, int id) : Person(n), employeeID(id) {}
```

```
    void display() const override {
        std::cout << "Employee: " << name << " (ID: " << employeeID << ")" <<
            std::endl;
    }
};

class Student : virtual public Person {
protected:
    int studentID;

public:
    Student(const std::string& n, int id) : Person(n), studentID(id) {}

    void display() const override {
        std::cout << "Student: " << name << " (ID: " << studentID << ")" << std::endl;
    }
};

class Manager : public Employee, public Student {
public:
    Manager(const std::string& n, int empID, int stdID)
        : Person(n), Employee(n, empID), Student(n, stdID) {}

    void display() const override {
        Employee::display();
        Student::display();
        std::cout << "Manager: " << name << " (Employee ID: " << employeeID << ",
            Student ID: " << studentID << ")" << std::endl;
    }
};

int main() {
    Manager manager("John Doe", 101, 202);

    // Display the information of the Manager
    manager.display();

    return 0;
}
```

Output:

```
Employee: John Doe (ID: 101)
Student: John Doe (ID: 202)
Manager: John Doe (Employee ID: 101, Student ID: 202)
```

3. Write a program with an abstract class `Student` and create derive classes `Engineering`, `Medicine` and `Science` from base class `Student`. Create the objects of the derived classes and process them and access them using an array of pointers of type base class `Student`.

```
#include <iostream>
#include <string>

// Abstract base class Student
class Student {
protected:
    std::string name;
    int rollNumber;

public:
    Student(const std::string& n, int roll) : name(n), rollNumber(roll) {}

    // Pure virtual function to be implemented by derived classes
    virtual void display() const = 0;
};

// Derived class Engineering
class Engineering : public Student {
private:
    std::string branch;

public:
    Engineering(const std::string& n, int roll, const std::string& b)
        : Student(n, roll), branch(b) {}

    void display() const override {
        std::cout << "Engineering Student - Name: " << name << ", Roll Number: " <<
            rollNumber << ", Branch: " << branch << std::endl;
    }
};

// Derived class Medicine
class Medicine : public Student {
private:
    std::string specialization;

public:
    Medicine(const std::string& n, int roll, const std::string& s)
        : Student(n, roll), specialization(s) {}

    void display() const override {
```

```
        std::cout << "Medicine Student - Name: " << name << ", Roll Number: " <<
            rollNumber << ", Specialization: " << specialization << std::endl;
    }
};

// Derived class Science
class Science : public Student {
private:
    std::string field;

public:
    Science(const std::string& n, int roll, const std::string& f)
        : Student(n, roll), field(f) {}

    void display() const override {
        std::cout << "Science Student - Name: " << name << ", Roll Number: " <<
            rollNumber << ", Field: " << field << std::endl;
    }
};

int main() {
    // Create an array of pointers to the base class Student
    const int numStudents = 3;
    Student* students[numStudents];

    // Create objects of derived classes and store them in the array
    students[0] = new Engineering("Alice", 101, "Computer Science");
    students[1] = new Medicine("Bob", 102, "Cardiology");
    students[2] = new Science("Charlie", 103, "Physics");

    // Process and display information using the base class pointers
    for (int i = 0; i < numStudents; ++i) {
        students[i]->display();
    }

    // Clean up allocated memory
    for (int i = 0; i < numStudents; ++i) {
        delete students[i];
    }

    return 0;
}
```

Output:
Engineering Student - Name: Alice, Roll Number: 101, Branch: Computer Science
Medicine Student - Name: Bob, Roll Number: 102, Specialization: Cardiology
Science Student - Name: Charlie, Roll Number: 103, Field: Physics

4. Create a polymorphic class `Vehicle` and create other derived classes `Bus`, `Car`, and `Bike` from `Vehicle`. Illustrate RTTI by the use of `dynamic_cast` and `typeid` operators in this program.

```
#include <iostream>
#include <typeinfo>

class Vehicle {
public:
    virtual void displayType() const {
        std::cout << "This is a Vehicle." << std::endl;
    }
};

class Bus : public Vehicle {
public:
    void displayType() const override {
        std::cout << "This is a Bus." << std::endl;
    }
};

class Car : public Vehicle {
public:
    void displayType() const override {
        std::cout << "This is a Car." << std::endl;
    }
};

class Bike : public Vehicle {
public:
    void displayType() const override {
        std::cout << "This is a Bike." << std::endl;
    }
};

int main() {
    Vehicle* vehicles[4];

    vehicles[0] = new Bus();
    vehicles[1] = new Car();
    vehicles[2] = new Bike();
```

```
    vehicles[3] = new Vehicle(); // A generic Vehicle

    for (int i = 0; i < 4; ++i) {
        // Using dynamic_cast to check the actual type
        if (Bus* bus = dynamic_cast<Bus*>(vehicles[i])) {
            std::cout << "Dynamic cast: Bus" << std::endl;
        } else if (Car* car = dynamic_cast<Car*>(vehicles[i])) {
            std::cout << "Dynamic cast: Car" << std::endl;
        } else if (Bike* bike = dynamic_cast<Bike*>(vehicles[i])) {
            std::cout << "Dynamic cast: Bike" << std::endl;
        } else {
            std::cout << "Dynamic cast: Vehicle" << std::endl;
        }

        // Using typeid operator to check the type
        if (typeid(*vehicles[i]) == typeid(Bus)) {
            std::cout << "Type ID: Bus" << std::endl;
        } else if (typeid(*vehicles[i]) == typeid(Car)) {
            std::cout << "Type ID: Car" << std::endl;
        } else if (typeid(*vehicles[i]) == typeid(Bike)) {
            std::cout << "Type ID: Bike" << std::endl;
        } else {
            std::cout << "Type ID: Vehicle" << std::endl;
        }

        vehicles[i]->displayType();
        delete vehicles[i];
    }

    return 0;
}
```

Output:

```
Dynamic cast: Bus
Type ID: Bus
This is a Bus.
Dynamic cast: Car
Type ID: Car
This is a Car.
Dynamic cast: Bike
Type ID: Bike
This is a Bike.
Dynamic cast: Vehicle
Type ID: Vehicle
This is a Vehicle.
```