Lab5(

**1.)/*Write a class for instantiating the objects that represent the two-dimensional
Cartesian coordinate system. A. Make a particular member function of one class
as a friend function of another class for addition. B. Make the other three functions to work as a bridge between the classes for multiplication, division, and subtraction. C. Also write a small program to demonstrate that all the member functions of one class are the friend functions of another class if the former class is made friend to the latter. Make least possible classes to demonstrate all the above in a single program without conflict.*/**

```
#include <iostream>
using namespace std;
class CartesianCoord;
class CoordBridge {
public:
  static CartesianCoord add(const CartesianCoord &coord1,
                            const CartesianCoord &coord2);
  static CartesianCoord subtract(const CartesianCoord &coord1,
                                 const CartesianCoord &coord2);
  static CartesianCoord multiply(const CartesianCoord &coord1,
                                 const CartesianCoord &coord2);
  static CartesianCoord divide(const CartesianCoord &coord1,
                               const CartesianCoord &coord2);
};
class CartesianCoord {
private:
  double x, y;

public:
  CartesianCoord(double x = 0, double y = 0) : x(x), y(y) {}
  friend CartesianCoord CoordBridge::add(const CartesianCoord &coord1,
                                         const CartesianCoord &coord2);
  friend CartesianCoord CoordBridge::subtract(const CartesianCoord &coord1,
                                              const CartesianCoord &coord2);
  friend CartesianCoord CoordBridge::multiply(const CartesianCoord &coord1,
                                              const CartesianCoord &coord2);
  friend CartesianCoord CoordBridge::divide(const CartesianCoord &coord1,
                                            const CartesianCoord &coord2);
  void display() const { cout << "(" << x << ", " << y << ")" << endl; }
};

CartesianCoord CoordBridge::add(const CartesianCoord &coord1,
                                const CartesianCoord &coord2) {
  CartesianCoord result;
  result.x = coord1.x + coord2.x;
  result.y = coord1.y + coord2.y;
```

```
  return result;
}

CartesianCoord CoordBridge::subtract(const CartesianCoord &coord1,
                                     const CartesianCoord &coord2) {
  CartesianCoord result;
  result.x = coord1.x - coord2.x;
  result.y = coord1.y - coord2.y;
  return result;
}

CartesianCoord CoordBridge::multiply(const CartesianCoord &coord1,
                                     const CartesianCoord &coord2) {
  CartesianCoord result;
  result.x = coord1.x * coord2.x;
  result.y = coord1.y * coord2.y;
  return result;
}

CartesianCoord CoordBridge::divide(const CartesianCoord &coord1,
                                   const CartesianCoord &coord2) {
  CartesianCoord result;
  result.x = coord1.x / coord2.x;
  result.y = coord1.y / coord2.y;
  return result;
}

int main() {
  CartesianCoord coord1(2, 3), coord2(4, 5);

  cout << "Coord1: ";
  coord1.display();
  cout << "Coord2: ";
  coord2.display();

  cout << "Addition: ";
  CartesianCoord sum = CoordBridge::add(coord1, coord2);
  sum.display();

  cout << "Subtraction: ";
  CartesianCoord diff = CoordBridge::subtract(coord1, coord2);
  diff.display();

  cout << "Multiplication: ";
  CartesianCoord prod = CoordBridge::multiply(coord1, coord2);
  prod.display();

  cout << "Division: ";
```

```
  CartesianCoord quot = CoordBridge::divide(coord1, coord2);
  quot.display();

  return 0;
}
```

Output

```
Coord1: (2, 3)
Coord2: (4, 5)
Addition: (6, 8)
Subtraction: (-2, -2)
Multiplication: (8, 15)
Division: (0.5, 0.6)
```

2.)

```
/*Write a class to store x, y, and z coordinates of a point in three-dimensional
 * space. Overload addition and subtraction operators for addition and
 * subtraction of two coordinate objects. Implement the operator functions as
 * non-member functions (friend operator functions).*/
#include <iostream>
using namespace std;
class Point3D {
private:
  double x, y, z;

public:
  Point3D(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}

  // Overloaded operators as friend functions
  friend Point3D operator+(const Point3D &lhs, const Point3D &rhs);
  friend Point3D operator-(const Point3D &lhs, const Point3D &rhs);

  // Display function to print the coordinates
  void display() const {
    cout << "(" << x << ", " << y << ", " << z << ")" << endl;
  }
};

// Overloaded addition operator
Point3D operator+(const Point3D &lhs, const Point3D &rhs) {
  Point3D result;
  result.x = lhs.x + rhs.x;
  result.y = lhs.y + rhs.y;
  result.z = lhs.z + rhs.z;
  return result;
}

// Overloaded subtraction operator
Point3D operator-(const Point3D &lhs, const Point3D &rhs) {
```

```
  Point3D result;
  result.x = lhs.x - rhs.x;
  result.y = lhs.y - rhs.y;
  result.z = lhs.z - rhs.z;
  return result;
}

int main() {
  Point3D p1(7, 8, 9), p2(2, 3, 4);

  cout << "p1: ";
  p1.display();
  cout << "p2: ";
  p2.display();

  cout << "Addition: ";
  Point3D sum = p1 + p2;
  sum.display();

  cout << "Subtraction: ";
  Point3D diff = p1 - p2;
  diff.display();

  return 0;
}
```

output :

```
p1: (7, 8, 9)
p2: (2, 3, 4)
Addition: (9, 11, 13)
Subtraction: (5, 5, 5)
```

3.)

/Write a program to compare two objects of a class that contains an integer value as its data member. Make overloading functions to overload equality(==), less than(<), greater than(>), not equal (!=), greater than or equal to (>=), and less than or equal to(<=) operators using member operator functions.*/

```
#include <iostream>
using namespace std;
class Integer {
private:
  int value;

public:
  Integer(int value = 0) : value(value) {}

  // Overloaded comparison operators
  bool operator==(const Integer &other) const { return value == other.value; }
```

```
  bool operator<(const Integer &other) const { return value < other.value; }

  bool operator>(const Integer &other) const { return value > other.value; }

  bool operator!=(const Integer &other) const { return !(*this == other); }

  bool operator>=(const Integer &other) const {
    return (*this > other) || (*this == other);
  }

  bool operator<=(const Integer &other) const {
    return (*this < other) || (*this == other);
  }

  // Display function to print the value
  void display() const { std::cout << value << std::endl; }
};

int main() {
  Integer a(5), b(10);

  cout << "a: ";
  a.display();
  cout << "b: ";
  b.display();

  cout << "a == b: " << (a == b) << endl;
  cout << "a < b: " << (a < b) << endl;
  cout << "a > b: " << (a > b) << endl;
  cout << "a != b: " << (a != b) << endl;
  cout << "a >= b: " << (a >= b) << endl;
  cout << "a <= b: " << (a <= b) << endl;

  return 0;
}
```

Output:

```
a: 5
b: 10
a == b: 0
a < b: 1
a > b: 0
a != b: 1
a >= b: 0
a <= b: 1
```

4.)/*Write a class Date that overloads prefix and postfix operators to increase the Date object by one day, while causing appropriate increments to the month and year (use the appropriate condition for leap year). The prefix and postfix operators in the Date class should behave exactly like the built-in increment operators.*/

```
#include <iostream>
using namespace std;
class Date
{
private:
   int day, month, year;
public:
   Date(int day = 1, int month = 1, int year = 2000) : day(day), month(month),
year(year) {}
   // Prefix increment operator
   Date &operator++()
   {
      // Check if it's the last day of the month
      int lastDay = 31;
      if (month == 2)
      {
         if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
         {
            lastDay = 29;
         }
         else
         {
            lastDay = 28;
         }
      }
      else if (month == 4 || month == 6 || month == 9 || month == 11)
      {
         lastDay = 30;
      }

      if (day == lastDay)
      {
         day = 1;
         if (month == 12)
         {
            month = 1;
            year++;
         }
         else
         {
            month++;
         }
      }
```

```
        else
        {
            day++;
        }

        return *this;
    }

    // Postfix increment operator
    Date operator++(int)
    {
        Date temp(*this);
        ++(*this);
        return temp;
    }

    // Display function to print the date
    void display() const
    {
        cout << day << "/" << month << "/" << year << endl;
    }
};
int main()
{
    Date d(14, 11, 2021);

    cout << "Original date: ";
    d.display();

    cout << "Prefix increment: ";
    (++d).display();

    cout << "Postfix increment: ";
    (d++).display();

    cout << "Final date: ";
    d.display();

    return 0;
}
```

Output:

Original date: 14/11/2021
Prefix increment: 15/11/2021
Postfix increment: 15/11/2021
Final date: 16/11/2021

Lab 6

1. **Write a program that can convert the Distance (meter, centimeter) to meters measurement in float and vice versa. Make a class distance with two data members, meter and centimeter. You can add function members as per your requirement.**

```
#include <iostream>

class Distance {
private:
    float meter;
    float centimeter;

public:
    // Constructor to initialize meter and centimeter
    Distance(float m, float cm) : meter(m), centimeter(cm) {}

    // Function to convert Distance to meters
    float toMeters() {
        return meter + centimeter / 100.0; // 1 meter = 100 centimeters
    }

    // Function to convert Distance to centimeters
    float toCentimeters() {
        return meter * 100.0 + centimeter;
    }
};

int main() {
    float m, cm;

    std::cout << "Enter distance in meters: ";
    std::cin >> m;

    std::cout << "Enter distance in centimeters: ";
    std::cin >> cm;

    // Create a Distance object
    Distance d(m, cm);

    std::cout << "Distance in meters: " << d.toMeters() << " meters" << std::endl;
    std::cout << "Distance in centimeters: " << d.toCentimeters() << " centimeters"
<< std::endl;

    return 0;
}
```

Output:
Enter distance in meters: 2
Enter distance in centimeters: 4
Distance in meters: 2.04 meters
Distance in centimeters: 204 centimeters

2. Write two classes to store distances in meter-centimeter and feet-inch systems respectively. Write conversions functions so that the program can convert objects of both types.

```
#include <iostream>

class DistanceFeetInch;

class DistanceMeterCentimeter {
private:
    float meter;
    float centimeter;

public:
    DistanceMeterCentimeter(float m, float cm) : meter(m), centimeter(cm) {}

    // Conversion function to convert to DistanceFeetInch
    DistanceFeetInch toFeetInch();

    void display() {
        std::cout << "Distance in meter-centimeter: " << meter << " meters " <<
centimeter << " centimeters" << std::endl;
    }
};

class DistanceFeetInch {
private:
    float feet;
    float inch;

public:
    DistanceFeetInch(float ft, float in) : feet(ft), inch(in) {}

    // Conversion function to convert to DistanceMeterCentimeter
    DistanceMeterCentimeter toMeterCentimeter();

    void display() {
        std::cout << "Distance in feet-inch: " << feet << " feet " << inch << " inches"
<< std::endl;
    }
};

// Conversion function from DistanceMeterCentimeter to DistanceFeetInch
```

```
DistanceFeetInch DistanceMeterCentimeter::toFeetInch() {
    float totalInches = (meter * 100 + centimeter) / 2.54; // 1 inch = 2.54 centimeters
    float feet = totalInches / 12;
    float inch = totalInches - feet * 12;
    return DistanceFeetInch(feet, inch);
}

// Conversion function from DistanceFeetInch to DistanceMeterCentimeter
DistanceMeterCentimeter DistanceFeetInch::toMeterCentimeter() {
    float totalInches = feet * 12 + inch;
    float cm = totalInches * 2.54; // 1 inch = 2.54 centimeters
    float meter = cm / 100;
    cm = cm - (static_cast<int>(meter) * 100); // Remaining centimeters
    return DistanceMeterCentimeter(meter, cm);
}

int main() {
    // Example using DistanceMeterCentimeter
    DistanceMeterCentimeter distanceMC(5, 30);
    distanceMC.display();
    DistanceFeetInch convertedFI = distanceMC.toFeetInch();
    convertedFI.display();

    // Example using DistanceFeetInch
    DistanceFeetInch distanceFI(2, 6);
    distanceFI.display();
    DistanceMeterCentimeter convertedMC = distanceFI.toMeterCentimeter();
    convertedMC.display();

    return 0;
}
```

Output

```
Distance in meter-centimeter: 5 meters 30 centimeters
Distance in feet-inch: 17.3885 feet -7.62939e-06 inches
Distance in feet-inch: 2 feet 6 inches
Distance in meter-centimeter: 0.762 meters 76.2 centimeters
```

3. Create a class called Musicians to contain three methods `string()`, `wind()`, and `perc()`. Each of these methods should initialize a string array to contain the following instruments

- **veena, guitar, sitar, sarod and mandolin under `string()`**
- **flute, clarinet saxophone, nadhaswaram, and piccolo under `wind()`**
- **tabla, mridangam, bangos, drums and tambour under `perc()`**

It should also display the contents of the arrays that are initialized. Create a derived class called `TypeIns` to contain a method

called `get()` and `show()`. The `get()` method must display a menu as follows

Type of instruments to be displayed
a. String instruments
b. Wind instruments
c. Percussion instruments

The `show()` method should display the relevant detail according to our choice. The base class variables must be accessible only to their derived classes.

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

class Musicians {
protected:
    std::string stringInstruments[5];
    std::string windInstruments[5];
    std::string percInstruments[5];

public:
    Musicians() {
        // Initialize the string array with string instruments
        stringInstruments[0] = "veena";
        stringInstruments[1] = "guitar";
        stringInstruments[2] = "sitar";
        stringInstruments[3] = "sarod";
        stringInstruments[4] = "mandolin";

        // Initialize the wind array with wind instruments
        windInstruments[0] = "flute";
        windInstruments[1] = "clarinet";
        windInstruments[2] = "saxophone";
        windInstruments[3] = "nadhaswaram";
        windInstruments[4] = "piccolo";

        // Initialize the perc array with percussion instruments
        percInstruments[0] = "tabla";
        percInstruments[1] = "mridangam";
        percInstruments[2] = "bongos";
        percInstruments[3] = "drums";
        percInstruments[4] = "tambour";
    }
```

```
    void displayStringInstruments() {
        std::cout << "String Instruments:" << std::endl;
        for (int i = 0; i < 5; i++) {
            std::cout << stringInstruments[i] << std::endl;
        }
    }

    void displayWindInstruments() {
        std::cout << "Wind Instruments:" << std::endl;
        for (int i = 0; i < 5; i++) {
            std::cout << windInstruments[i] << std::endl;
        }
    }

    void displayPercInstruments() {
        std::cout << "Percussion Instruments:" << std::endl;
        for (int i = 0; i < 5; i++) {
            std::cout << percInstruments[i] << std::endl;
        }
    }
};

class TypeIns : public Musicians {
public:
    void get() {
        char choice;
        std::cout << "Type of instruments to be displayed:" << std::endl;
        std::cout << "a. String instruments" << std::endl;
        std::cout << "b. Wind instruments" << std::endl;
        std::cout << "c. Percussion instruments" << std::endl;
        std::cout << "Enter your choice: ";
        std::cin >> choice;

        switch (choice) {
            case 'a':
                displayStringInstruments();
                break;
            case 'b':
                displayWindInstruments();
                break;
            case 'c':
```

```
            displayPercInstruments();
            break;
        default:
            std::cout << "Invalid choice" << std::endl;
      }
   }

   void show() {
      get();
   }
};

int main() {
   TypeIns typeIns;
   typeIns.show();
   return 0;
}
```

Output:

```
Type of instruments to be displayed:
a. String instruments
b. Wind instruments
c. Percussion instruments
Enter your choice: a
String Instruments:
veena
guitar
sitar
sarod
mandolin
Enter your choice: b
Wind Instruments:
flute
clarinet
saxophone
nadhaswaram
piccolo
Enter your choice: c
Percussion Instruments:
tabla
mridangam
bongos
```

drums
tambour

4. Write three derived classes inheriting functionality of base class `person` (should have a member function that asks to enter name and age) and with added unique features of `student`, and `employee`, and functionality to assign, change and delete records of student and employee. And make one member function for printing the address of the objects of classes (base and derived) using this pointer. Create two objects of the base class and derived classes each and print the addresses of individual objects. Using a calculator, calculate the address space occupied by each object and verify this with address spaces printed by the program.

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

class Person {
protected:
    std::string name;
    int age;

public:
    Person() : name(""), age(0) {}

    void enterData() {
        std::cout << "Enter name: ";
        std::cin >> name;
        std::cout << "Enter age: ";
        std::cin >> age;
    }

    void printAddress() {
        std::cout << "Address of Person object: " << this << std::endl;
    }
};

class Student : public Person {
private:
    int studentID;

public:
```

```
    Student() : studentID(0) {}

    void assignStudentRecord(int id) {
        studentID = id;
    }

    void changeStudentRecord(int id) {
        studentID = id;
    }

    void deleteStudentRecord() {
        studentID = 0;
    }

    void printAddress() {
        std::cout << "Address of Student object: " << this << std::endl;
    }
};

class Employee : public Person {
private:
    int employeeID;

public:
    Employee() : employeeID(0) {}

    void assignEmployeeRecord(int id) {
        employeeID = id;
    }

    void changeEmployeeRecord(int id) {
        employeeID = id;
    }

    void deleteEmployeeRecord() {
        employeeID = 0;
    }

    void printAddress() {
        std::cout << "Address of Employee object: " << this << std::endl;
    }
};
```

```
int main() {
    Person person1;
    Person person2;
    Student student1;
    Student student2;
    Employee employee1;
    Employee employee2;

    person1.enterData();
    person2.enterData();
    student1.enterData();
    student2.enterData();
    employee1.enterData();
    employee2.enterData();

    person1.printAddress();
    person2.printAddress();
    student1.printAddress();
    student2.printAddress();
    employee1.printAddress();
    employee2.printAddress();

    // Calculate address space occupied by objects
    std::cout << "Size of Person object: " << sizeof(Person) << " bytes" <<
        std::endl;
    std::cout << "Size of Student object: " << sizeof(Student) << " bytes" <<
        std::endl;
    std::cout << "Size of Employee object: " << sizeof(Employee) << "
        bytes" << std::endl;

    return 0;
}
```

Output

```
Enter name: Ramesh
Enter age: 21
Enter name: Hari
Enter age: 23
Enter name: Santosh
Enter age: 21
Enter name: Srinivash
Enter age: 22
```

```
Enter name: Swami
Enter age: 21
Enter name: Satyam
Enter age: 21
Address of Person object: 0x16b1e2fe8
Address of Person object: 0x16b1e2fc8
Address of Student object: 0x16b1e2f98
Address of Student object: 0x16b1e2f78
Address of Employee object: 0x16b1e2f58
Address of Employee object: 0x16b1e2f38
Size of Person object: 32 bytes
Size of Student object: 32 bytes
Size of Employee object: 32 bytes
```

5. Write a base class that asks the user to enter a complex number and make a derived class that adds the complex number of its own with the base. Finally, make a third class that is a friend of derived and calculate the difference of the base complex number and its own complex number.

```
#include <iostream>

class Complex {
protected:
    double real;
    double imag;

public:
    Complex() : real(0.0), imag(0.0) {}

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

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

    Complex operator-(const Complex& other) const {
        return Complex(real - other.real, imag - other.imag);
    }

    void enterComplexNumber() {
        std::cout << "Enter real part: ";
```

```
        std::cin >> real;
        std::cout << "Enter imaginary part: ";
        std::cin >> imag;
    }

    void displayComplexNumber() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};

int main() {
    Complex complex1;
    Complex complex2;

    std::cout << "Enter the first complex number:" << std::endl;
    complex1.enterComplexNumber();

    std::cout << "Enter the second complex number:" << std::endl;
    complex2.enterComplexNumber();

    Complex resultAddition = complex1 + complex2;
    Complex resultSubtraction = complex1 - complex2;

    std::cout << "Result of addition: ";
    resultAddition.displayComplexNumber();

    std::cout << "Result of subtraction: ";
    resultSubtraction.displayComplexNumber();

    return 0;
}
```

Output:

```
Enter the first complex number:
Enter real part: 1
Enter imaginary part: 2
Enter the second complex number:
Enter real part: 1
Enter imaginary part: 2
Result of addition: 2 + 4i
Result of subtraction: 0 + 0i
```

Lab 7

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.
```

Lab 8

1. Write a program to demonstrate the use of different `ios` flags and functions to format the output. Create a program to generate the bill invoice of a department store by using different formatting.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>

// Define a struct to represent an item in the invoice
struct InvoiceItem {
    std::string name;
    int quantity;
    double price;
};

// Function to generate and display the bill invoice
void generateInvoice(const std::vector<InvoiceItem>& items) {
    // Set up the header
     std::cout << std::left << std::setw(20) << "Item Name" << std::setw(10)
<< "Quantity" << std::setw(10) << "Price" << std::setw(15) << "Total" <<
std::endl;
      std::cout << std::setfill('-') << std::setw(55) << "" << std::setfill(' ') <<
std::endl;

    // Calculate and display the items and totals
    double totalAmount = 0.0;
    for (const auto& item : items) {
        double itemTotal = item.quantity * item.price;
         std::cout << std::left << std::setw(20) << item.name << std::setw(10)
<< item.quantity << std::fixed << std::setprecision(2) << std::setw(10) <<
item.price << std::setw(15) << itemTotal << std::endl;
        totalAmount += itemTotal;
    }

    // Display the total amount
      std::cout << std::setfill('-') << std::setw(55) << "" << std::setfill(' ') <<
std::endl;
          std::cout << std::setw(45) << "Total Amount" << std::fixed <<
std::setprecision(2) << std::setw(10) << totalAmount << std::endl;
}
```

```
int main() {
    // Create sample invoice items
    std::vector<InvoiceItem> items;
    items.push_back({"Item 1", 3, 10.50});
    items.push_back({"Item 2", 2, 25.75});
    items.push_back({"Item 3", 5, 5.99});

    // Display the bill invoice with different formatting
    std::cout << "Default Formatting:" << std::endl;
    generateInvoice(items);

    std::cout << "\nUsing Fixed Notation:" << std::endl;
    std::cout << std::fixed;
    generateInvoice(items);

    std::cout << "\nUsing Scientific Notation:" << std::endl;
    std::cout << std::scientific;
    generateInvoice(items);

    return 0;
}
```

Output:

```
Default Formatting:
Item Name          Quantity  Price     Total
-------------------------------------------------------
Item 1             3         10.50     31.50
Item 2             2         25.75     51.50
Item 3             5         5.99      29.95
-------------------------------------------------------
Total Amount                             112.95

Using Fixed Notation:
Item Name          Quantity  Price     Total
-------------------------------------------------------
Item 1             3         10.50     31.50
Item 2             2         25.75     51.50
Item 3             5         5.99      29.95
-------------------------------------------------------
Total Amount                             112.95
```

```
Using Scientific Notation:
Item Name          Quantity  Price     Total
-----------------------------------------------------
Item 1             3         10.50     31.50
Item 2             2         25.75     51.50
Item 3             5         5.99      29.95
-----------------------------------------------------
Total Amount                           112.95
```

2. Write a program to create a user-defined manipulator that will format the output by setting the width, precision, and fill character at the same time by passing arguments.

```
#include <iostream>
#include <iomanip>

// User-defined manipulator
struct FormatManipulator {
    int width;
    int precision;
    char fill;

    FormatManipulator(int w, int p, char f) : width(w), precision(p), fill(f) {}
};

// Overload the << operator for the user-defined manipulator
std::ostream& operator<<(std::ostream& os, const FormatManipulator&
manipulator) {
    os.width(manipulator.width);
    os.precision(manipulator.precision);
    os.fill(manipulator.fill);
    return os;
}

// Example usage
int main() {
    double number = 3.14159;

    std::cout << "Default output: " << number << std::endl;

    std::cout << "Formatted output: "
              << FormatManipulator(10, 4, '*') << number << std::endl;
```

```
    return 0;
}
```

Output:

Default output: 3.14159
Formatted output: *****3.142

3. Write a program to overload stream operators to read a complex number and display the complex number in a+ib format.

```
#include <iostream>

class Complex {
private:
    double real;
    double imaginary;

public:
    Complex(double real = 0.0, double imaginary = 0.0)
        : real(real), imaginary(imaginary) {}

    friend std::istream& operator>>(std::istream& in, Complex& complex);
       friend std::ostream& operator<<(std::ostream& out, const Complex&
complex);
};

std::istream& operator>>(std::istream& in, Complex& complex) {
    std::cout << "Enter the real part: ";
    in >> complex.real;

    std::cout << "Enter the imaginary part: ";
    in >> complex.imaginary;

    return in;
}

std::ostream& operator<<(std::ostream& out, const Complex& complex) {
    out << complex.real;
    if (complex.imaginary >= 0)
        out << "+";
    out << complex.imaginary << "i";

    return out;
}
```

```
int main() {
    Complex c;

    std::cout << "Enter a complex number:" << std::endl;
    std::cin >> c;

    std::cout << "Complex number in a+ib format: " << c << std::endl;

    return 0;
}
```

Output:

```
Enter a complex number:
Enter the real part: 1
Enter the imaginary part: 2
Complex number in a+ib format: 1+2i
```

4. Write a program that stores the information about students (name, student id, department, and address) in a structure and then transfers the information to a file in your directory. Finally, retrieve the information from your file and print it in the proper format on your output screen.

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

// Define a structure to store student information
struct Student {
    std::string name;
    int studentId;
    std::string department;
    std::string address;
};

// Function to write student information to a file
void writeStudentToFile(const Student& student, const std::string& filename) {
     std::ofstream outFile(filename, std::ios::app); // Open the file in append mode
    if (!outFile) {
        std::cerr << "Error opening the file for writing." << std::endl;
        return;
```

```
    }

    // Write student information to the file
    outFile << "Name: " << student.name << std::endl;
    outFile << "Student ID: " << student.studentId << std::endl;
    outFile << "Department: " << student.department << std::endl;
    outFile << "Address: " << student.address << std::endl;
    outFile << std::endl;

    outFile.close();
}

// Function to read and print student information from a file
void readStudentFromFile(const std::string& filename) {
    std::ifstream inFile(filename);
    if (!inFile) {
        std::cerr << "Error opening the file for reading." << std::endl;
        return;
    }

    std::string line;
    while (std::getline(inFile, line)) {
        std::cout << line << std::endl;
    }

    inFile.close();
}

int main() {
    // Create and initialize a student structure
     Student student1 = {"John Doe", 101, "Computer Science", "123 Main
St"};
     Student student2 = {"Jane Smith", 102, "Electrical Engineering", "456
Elm St"};

    // Write student information to a file
    writeStudentToFile(student1, "student_info.txt");
    writeStudentToFile(student2, "student_info.txt");

    // Read and print student information from the file
    std::cout << "Student Information from File:" << std::endl;
    readStudentFromFile("student_info.txt");
```

```
    return 0;
}
```

Output:

```
Name: Ram Joshi
Student ID: 101
Department: Computer Science
Address: 123 Main St

Name: Hari Yadav
Student ID: 102
Department: Electrical Engineering
Address: 456 Elm St

Name: Ram Joshi
Student ID: 101
Department: Computer Science
Address: 123 Main St

Name: Hari Yadav
Student ID: 102
Department: Electrical Engineering
Address: 456 Elm St
```

5. Write a program for transaction processing that write and read object randomly to and from a random access file so that user can add, update, delete and display the account information (account-number, last-name, first-name, total-balance).

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

// Define the structure for account information
struct Account {
    int accountNumber;
    std::string lastName;
    std::string firstName;
    double totalBalance;
};
```

```
// Function to display an account's details
void displayAccount(const Account& account) {
        std::cout << "Account Number: " << account.accountNumber <<
std::endl;
    std::cout << "Last Name: " << account.lastName << std::endl;
    std::cout << "First Name: " << account.firstName << std::endl;
    std::cout << "Total Balance: $" << account.totalBalance << std::endl;
    std::cout << "----------------------" << std::endl;
}

// Function to add a new account to the file
void addAccount(std::fstream& file, const Account& account) {
    file.write(reinterpret_cast<const char*>(&account), sizeof(Account));
}

// Function to update an account in the file
void updateAccount(std::fstream& file, int accountNumber, const
Account& updatedAccount) {
    Account account;
    while (file.read(reinterpret_cast<char*>(&account), sizeof(Account))) {
        if (account.accountNumber == accountNumber) {
                    file.seekp(-static_cast<std::streamoff>(sizeof(Account)),
std::ios::cur);
                file.write(reinterpret_cast<const char*>(&updatedAccount),
sizeof(Account));
            break;
        }
    }
}

// Function to delete an account from the file
void deleteAccount(std::fstream& file, int accountNumber) {
    std::fstream tempFile("temp.txt", std::ios::out | std::ios::binary);
    if (!tempFile) {
        std::cerr << "Error creating temporary file." << std::endl;
        return;
    }

    Account account;
    while (file.read(reinterpret_cast<char*>(&account), sizeof(Account))) {
        if (account.accountNumber != accountNumber) {
```

```
                    tempFile.write(reinterpret_cast<const char*>(&account),
sizeof(Account));
        }
    }

    file.close();
    tempFile.close();
    remove("accounts.txt");
    rename("temp.txt", "accounts.txt");

    file.open("accounts.txt", std::ios::in | std::ios::out | std::ios::binary);
}

// Function to display all accounts in the file
void displayAllAccounts(std::fstream& file) {
    file.seekg(0, std::ios::beg);
    Account account;
    while (file.read(reinterpret_cast<char*>(&account), sizeof(Account))) {
        displayAccount(account);
    }
}

int main() {
            std::fstream file("accounts.txt", std::ios::in | std::ios::out |
std::ios::binary);
    if (!file) {
        std::cerr << "Error opening the file." << std::endl;
        return 1;
    }

    int choice;
    do {
              std::cout << "1. Add Account\n2. Update Account\n3. Delete
Account\n4. Display All Accounts\n5. Exit\n";
        std::cout << "Enter your choice: ";
        std::cin >> choice;

        switch (choice) {
            case 1:
                {
                    Account newAccount;
                    std::cout << "Enter Account Number: ";
```

```
            std::cin >> newAccount.accountNumber;
            std::cout << "Enter Last Name: ";
            std::cin >> newAccount.lastName;
            std::cout << "Enter First Name: ";
            std::cin >> newAccount.firstName;
            std::cout << "Enter Total Balance: ";
            std::cin >> newAccount.totalBalance;
            addAccount(file, newAccount);
            std::cout << "Account added successfully." << std::endl;
        }
        break;
    case 2:
        {
            int accountNumber;
            std::cout << "Enter Account Number to Update: ";
            std::cin >> accountNumber;
            Account updatedAccount;
                    std::cout << "Enter Updated Account Information:" <<
std::endl;
            std::cout << "Enter Last Name: ";
            std::cin >> updatedAccount.lastName;
            std::cout << "Enter First Name: ";
            std::cin >> updatedAccount.firstName;
            std::cout << "Enter Total Balance: ";
            std::cin >> updatedAccount.totalBalance;
            updateAccount(file, accountNumber, updatedAccount);
            std::cout << "Account updated successfully." << std::endl;
        }
        break;
    case 3:
        {
            int accountNumber;
            std::cout << "Enter Account Number to Delete: ";
            std::cin >> accountNumber;
            deleteAccount(file, accountNumber);
            std::cout << "Account deleted successfully." << std::endl;
        }
        break;
    case 4:
        std::cout << "All Accounts:" << std::endl;
        displayAllAccounts(file);
        break;
```

```
            case 5:
                std::cout << "Exiting..." << std::endl;
                break;
            default:
                std::cout << "Invalid choice. Please try again." << std::endl;
        }
    } while (choice != 5);

    file.close();
    return 0;
}
```

Output

```
1. Add Account
2. Update Account
3. Delete Account
4. Display All Accounts
5. Exit
Enter your choice: 4
All Accounts:
Account Number: 123
Last Name: Frank
First Name: John
Total Balance: $3000
-----------------------
1. Add Account
2. Update Account
3. Delete Account
4. Display All Accounts
5. Exit
Enter your choice:
```

Lab 9(Understanding the Concept of Templates and Exception

1. **Create a function called `sum()` that returns the sum of the elements of an array. Make this function into a template so it will work with any numerical type. Write a program that applies this function to data of various types.**

```
#include <iostream>

// Templated function to calculate the sum of elements in an array
template <typename T>
T sum(const T array[], int size) {
    T result = 0;
```

```
    for (int i = 0; i < size; ++i) {
        result += array[i];
    }
    return result;
}

int main() {
    // Example 1: Integer array
    int intArray[] = {1, 2, 3, 4, 5};
    int intSum = sum(intArray, 5);
    std::cout << "Sum of integers: " << intSum << std::endl;

    // Example 2: Double array
    double doubleArray[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    double doubleSum = sum(doubleArray, 5);
    std::cout << "Sum of doubles: " << doubleSum << std::endl;

    // Example 3: Float array
    float floatArray[] = {0.5f, 1.5f, 2.5f, 3.5f, 4.5f};
    float floatSum = sum(floatArray, 5);
    std::cout << "Sum of floats: " << floatSum << std::endl;

    // Example 4: Long array
    long longArray[] = {1000, 2000, 3000, 4000, 5000};
    long longSum = sum(longArray, 5);
    std::cout << "Sum of longs: " << longSum << std::endl;

    return 0;
}
```

Output:
Sum of integers: 15
Sum of doubles: 16.5
Sum of floats: 12.5
Sum of longs: 15000

2. Write a class template for queue class. Assume the programmer using the queue won't make mistakes, like exceeding the capacity of the queue or trying to remove an item when the queue is empty. Define several queues of different data types and insert and remove data from them.

```
#include <iostream>
#include <vector>

// Define a class template for a queue
template <typename T>
```

```
class Queue {
public:
    // Constructor to initialize the queue
    Queue() {}

    // Function to insert an element at the rear of the queue
    void enqueue(const T& item) {
        elements.push_back(item);
    }

    // Function to remove and return an element from the front of the queue
    T dequeue() {
        if (!isEmpty()) {
            T frontElement = elements.front();
            elements.erase(elements.begin());
            return frontElement;
        }
        throw std::runtime_error("Queue is empty.");
    }

    // Function to check if the queue is empty
    bool isEmpty() const {
        return elements.empty();
    }

private:
    std::vector<T> elements;
};

int main() {
    // Create queues of different data types
    Queue<int> intQueue;
    Queue<double> doubleQueue;
    Queue<std::string> stringQueue;

    // Insert data into the queues
    intQueue.enqueue(10);
    intQueue.enqueue(20);
    intQueue.enqueue(30);

    doubleQueue.enqueue(3.14);
    doubleQueue.enqueue(2.718);

    stringQueue.enqueue("Hello");
    stringQueue.enqueue("World");
```

```
    // Remove and display data from the queues
    while (!intQueue.isEmpty()) {
        std::cout << "Dequeue int: " << intQueue.dequeue() << std::endl;
    }

    while (!doubleQueue.isEmpty()) {
        std::cout << "Dequeue double: " << doubleQueue.dequeue() << std::endl;
    }

    while (!stringQueue.isEmpty()) {
        std::cout << "Dequeue string: " << stringQueue.dequeue() << std::endl;
    }

    return 0;
}
```

Output:

```
Dequeue int: 10
Dequeue int: 20
Dequeue int: 30
Dequeue double: 3.14
Dequeue double: 2.718
Dequeue string: Hello
Dequeue string: World
```

4. Write any program that demonstrates the use of multiple catch handling, re-throwing an exception, and catching all exceptions.

```
#include <iostream>
#include <stdexcept>

// Function that throws different types of exceptions
void throwException(int choice) {
    if (choice == 1) {
        throw std::runtime_error("Runtime error occurred.");
    } else if (choice == 2) {
        throw std::logic_error("Logic error occurred.");
    } else if (choice == 3) {
        throw std::out_of_range("Out of range error occurred.");
    }
}

int main() {
    try {
        int choice;
```

```
        std::cout << "Enter 1 for runtime error, 2 for logic error, 3 for out of range error:
";
        std::cin >> choice;

        throwException(choice);
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught runtime_error: " << e.what() << std::endl;
    } catch (const std::logic_error& e) {
        std::cerr << "Caught logic_error: " << e.what() << std::endl;
    } catch (const std::out_of_range& e) {
        std::cerr << "Caught out_of_range: " << e.what() << std::endl;
    } catch (...) {
        // Catch all other exceptions
        std::cerr << "Caught an unknown exception." << std::endl;
    }

    return 0;
}
```

Output:

Enter 1 for runtime error, 2 for logic error, 3 for out of range error: 2
Caught logic_error: Logic error occurred.