1. Create a class hierarchy for vehicles, with a base class Vehicle and derived classes like Car, Truck, and Motorcycle. Implement file stream operations to read and write vehicle information to a text file, including details like vehicle type, model, and year.

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

// Base class
class Vehicle {
protected:
string type;
string model;
int year;
public:
Vehicle(string t, string m, int y) : type(t), model(m), year(y) {}
virtual void display() {
cout << "Type: " << type << ", Model: " << model << ", Year: " << year <<
endl;
}
virtual string toString() {
return type + "," + model + "," + to_string(year);
}
};

// Derived classes
class Car : public Vehicle {
public:
Car(string m, int y) : Vehicle("Car", m, y) {}
};

class Truck : public Vehicle {
public:
Truck(string m, int y) : Vehicle("Truck", m, y) {}
};

class Motorcycle : public Vehicle {
public:
Motorcycle(string m, int y) : Vehicle("Motorcycle", m, y) {}
};

// Function to write vehicles to file
void writeToFile(const vector<Vehicle*>& vehicles, const string& filename) {
ofstream file(filename);
if (!file) {
cerr << "Error opening file for writing!" << endl;
return;
}
for (auto v : vehicles) {
```

```
file << v->toString() << endl;
}
file.close();
}

// Function to read vehicles from file
void readFromFile(const string& filename) {
ifstream file(filename);
if (!file) {
cerr << "Error opening file for reading!" << endl;
return;
}
string type, model;
int year;
while (getline(file, type, ',') && getline(file, model, ',') && (file >> year)) {
file.ignore();
cout << "Type: " << type << ", Model: " << model << ", Year: " << year <<
endl;
}
file.close();
}
int main() {
vector<Vehicle*> vehicles = {
new Car("Toyota", 2020),
new Truck("Ford", 2018),
new Motorcycle("Honda", 2021)
};

string filename = "vehicles.txt";
writeToFile(vehicles, filename);

cout << "Vehicle details read from file:" << endl;
readFromFile(filename);

for (auto v : vehicles) delete v;
return 0;
}
```

2. Develop a c++ program for managing student records where you use text file handling to store student information such as name, ID, and grades in a text file. Additionally, implement binary file handling to store sensitive data like passwords securely.

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

// Structure for student record
struct Student {
   string name;
```

```
    int id;
    float grade;
};

// Function to add student record to text file
void addStudentRecord(const string& filename) {
    ofstream file(filename, ios::app);
    if (!file) {
        cerr << "Error opening file!" << endl;
        return;
    }
    Student s;
    cout << "Enter student name: ";
    cin.ignore();
    getline(cin, s.name);
    cout << "Enter student ID: ";
    cin >> s.id;
    cout << "Enter student grade: ";
    cin >> s.grade;
    file << s.name << "," << s.id << "," << s.grade << endl;
    file.close();
}

// Function to display student records
void displayStudentRecords(const string& filename) {
    ifstream file(filename);
    if (!file) {
        cerr << "Error opening file!" << endl;
        return;
    }
    string name;
    int id;
    float grade;
    while (getline(file, name, ',') && file >> id && file.ignore() && file >> grade) {
        file.ignore();
        cout << "Name: " << name << ", ID: " << id << ", Grade: " << grade << endl;
    }
    file.close();
}

// Function to store password securely in binary file
void storePassword(const string& filename) {
    ofstream file(filename, ios::binary | ios::app);
    if (!file) {
        cerr << "Error opening file!" << endl;
        return;
    }
    string password;
```

```
    cout << "Enter password to store: ";
    cin >> password;
    file.write(password.c_str(), password.size());
    file.close();
}

// Function to retrieve password from binary file
void retrievePasswords(const string& filename) {
    ifstream file(filename, ios::binary);
    if (!file) {
        cerr << "Error opening file!" << endl;
        return;
    }
    string password;
    while (!file.eof()) {
        char ch;
        file.get(ch);
        if (file.eof()) break;
        cout << ch;
    }
    cout << endl;
    file.close();
}

int main() {
    string studentFile = "students.txt";
    string passwordFile = "passwords.dat";
    int choice;
    do {
        cout << "\n1. Add Student Record";
        cout << "\n2. Display Student Records";
        cout << "\n3. Store Password";
        cout << "\n4. Retrieve Passwords";
        cout << "\n5. Exit";
        cout << "\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: addStudentRecord(studentFile); break;
            case 2: displayStudentRecords(studentFile); break;
            case 3: storePassword(passwordFile); break;
            case 4: retrievePasswords(passwordFile); break;
            case 5: cout << "Exiting program...\n"; break;
            default: cout << "Invalid choice!\n";
        }
    } while (choice != 5);
    return 0;
}
```

3. Write a program that reads data from a CSV file and calculates statistics such as average, maximum, and minimum values. Implement error handling to deal with file not found or data format errors using exception handling techniques like try-catch blocks.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <limits>
using namespace std;

// Function to read data from CSV and calculate statistics
void processCSV(const string& filename) {
    ifstream file;
    try {
        file.open(filename);
        if (!file.is_open()) {
            throw runtime_error("Error: File not found!");
        }

        vector<double> values;
        string line;
        while (getline(file, line)) {
            stringstream ss(line);
            string valueStr;
            while (getline(ss, valueStr, ',')) {
                try {
                    double value = stod(valueStr);
                    values.push_back(value);
                } catch (const invalid_argument&) {
                    cerr << "Warning: Invalid data format, skipping entry: " << valueStr << endl;
                }
            }
        }
        file.close();

        if (values.empty()) {
            throw runtime_error("Error: No valid numerical data found!");
        }

        double sum = 0, maxVal = numeric_limits<double>::min(), minVal =
numeric_limits<double>::max();
        for (double v : values) {
            sum += v;
            if (v > maxVal) maxVal = v;
            if (v < minVal) minVal = v;
        }

        cout << "Statistics:" << endl;
```

```
        cout << "Average: " << sum / values.size() << endl;
        cout << "Maximum: " << maxVal << endl;
        cout << "Minimum: " << minVal << endl;
    }
    catch (const exception& e) {
        cerr << e.what() << endl;
    }
}

int main() {
    string filename;
    cout << "Enter CSV filename: ";
    cin >> filename;
    processCSV(filename);
    return 0;
}
```

4. Create a hierarchy of shapes with a base class Shape and derived classes like Circle, Square, and Triangle. Use file stream operations to save and load shape data to/from a text file, including attributes like dimensions and colors.

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

// Base class
class Shape {
protected:
    string color;
public:
    Shape(string c) : color(c) {}
    virtual void display() = 0;
    virtual string toString() = 0;
};

// Derived class Circle
class Circle : public Shape {
    double radius;
public:
    Circle(string c, double r) : Shape(c), radius(r) {}
    void display() override {
        cout << "Circle: Color=" << color << ", Radius=" << radius << endl;
    }
    string toString() override {
        return "Circle," + color + "," + to_string(radius);
    }
};
```

```
// Derived class Square
class Square : public Shape {
   double side;
public:
   Square(string c, double s) : Shape(c), side(s) {}
   void display() override {
      cout << "Square: Color=" << color << ", Side=" << side << endl;
   }
   string toString() override {
      return "Square," + color + "," + to_string(side);
   }
};

// Derived class Triangle
class Triangle : public Shape {
   double base, height;
public:
   Triangle(string c, double b, double h) : Shape(c), base(b), height(h) {}
   void display() override {
      cout << "Triangle: Color=" << color << ", Base=" << base << ", Height=" << height <<
endl;
   }
   string toString() override {
      return "Triangle," + color + "," + to_string(base) + "," + to_string(height);
   }
};

// Function to write shape data to a file
void writeToFile(const vector<Shape*>& shapes, const string& filename) {
   ofstream file(filename);
   if (!file) {
      cerr << "Error opening file for writing!" << endl;
      return;
   }
   for (auto shape : shapes) {
      file << shape->toString() << endl;
   }
   file.close();
}

// Function to read shape data from a file
void readFromFile(const string& filename) {
   ifstream file(filename);
   if (!file) {
      cerr << "Error opening file for reading!" << endl;
      return;
   }
   string type, color;
```

```
    double dim1, dim2;
    while (getline(file, type, ',')) {
        getline(file, color, ',');
        if (type == "Circle") {
            file >> dim1;
            file.ignore();
            cout << "Circle: Color=" << color << ", Radius=" << dim1 << endl;
        } else if (type == "Square") {
            file >> dim1;
            file.ignore();
            cout << "Square: Color=" << color << ", Side=" << dim1 << endl;
        } else if (type == "Triangle") {
            file >> dim1;
            file.ignore();
            file >> dim2;
            file.ignore();
            cout << "Triangle: Color=" << color << ", Base=" << dim1 << ", Height=" << dim2 <<
endl;
        }
    }
    file.close();
}

int main() {
    vector<Shape*> shapes = {
        new Circle("Red", 5.5),
        new Square("Blue", 4.0),
        new Triangle("Green", 3.0, 6.0)
    };

    string filename = "shapes.txt";
    writeToFile(shapes, filename);

    cout << "Shapes read from file:" << endl;
    readFromFile(filename);

    for (auto shape : shapes) delete shape;
    return 0;
}
```

5. Write a program that reads student scores from a text file and calculates their average. Implement error handling to handle cases like file not found or invalid data format using exception handling, displaying a user-friendly message in case of errors

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <stdexcept>
```

```
using namespace std;

// Function to read student scores and calculate the average
void processScores(const string& filename) {
    ifstream file;
    try {
        file.open(filename);
        if (!file.is_open()) {
            throw runtime_error("Error: File not found!");
        }

        vector<double> scores;
        string line;
        while (getline(file, line)) {
            stringstream ss(line);
            string scoreStr;
            while (getline(ss, scoreStr, ',')) {
                try {
                    double score = stod(scoreStr);
                    scores.push_back(score);
                } catch (const invalid_argument&) {
                    cerr << "Warning: Invalid data format, skipping entry: " << scoreStr << endl;
                }
            }
        }
        file.close();

        if (scores.empty()) {
            throw runtime_error("Error: No valid numerical data found!");
        }

        double sum = 0;
        for (double s : scores) {
            sum += s;
        }
        cout << "Average Score: " << sum / scores.size() << endl;
    }
    catch (const exception& e) {
        cerr << e.what() << endl;
    }
}
int main() {
    string filename;
    cout << "Enter the student scores file name: ";
    cin >> filename;
    processScores(filename);
    return 0;
}
```