SWIFT LOGISTICS SYSTEM FUNCTIONAL REQUIREMENTS BREAKDOWN

1. Class Definitions

The program defines several classes, each with distinct responsibilities:

- **Vehicle (Base class)**: Defines common properties and behaviors for all vehicle types (e.g., type, model, registration number).
- **Car and Truck (Derived classes)**: Inherit from Vehicle and add specific attributes like seating capacity and payload capacity.
- **Customer**: Represents customer information, including name and contact details.

Encapsulation:

- Private members like name, contact, model, registrationNumber, etc., are encapsulated to restrict direct access from outside the class. The public functions (display(), saveToFile(), loadFromFile()) provide controlled access to these attributes.

Constructors and Destructors:

- Constructors are defined to initialize the objects with required data.
- The destructor in the Vehicle class ensures proper cleanup of dynamic memory when objects of derived classes are deleted.

Example:

```
class Vehicle {
protected:
   string type;
   string model;
   string registrationNumber;

public:
   Vehicle(string t, string m, string reg) : type(t), model(m), registrationNumber(reg) {}
   virtual ~Vehicle() {}
};
```

2. Inheritance

- **Vehicle Class (Base Class)**: Defines general vehicle-related behaviors and attributes.

- **Car and Truck (Derived Classes)**: Both Car and Truck inherit from Vehicle, demonstrating shared behaviors (e.g., display(), saveToFile()) while also adding specific functionality for each (seating capacity for cars and payload capacity for trucks).

Example of inheritance:

```
class Car : public Vehicle {
private:
   int seatingCapacity;
public:
   Car(string m, string reg, int sc) : Vehicle("Car", m, reg), seatingCapacity(sc) {}
   void display() const override {
      cout << "Car details...\n";
   }
};
```

3. Polymorphism

The program demonstrates **polymorphism** by using **virtual functions**. For instance, the display() function is overridden in both the Car and Truck classes, allowing the program to call the appropriate display() method for each specific vehicle type at runtime.

- The Vehicle class has a virtual display() method, and both Car and Truck override it to show their specific details.

Example:

```
class Vehicle {
public:
   virtual void display() const {
      cout << "Vehicle details...\n";
   }
};


class Car : public Vehicle {
public:
   void display() const override {
```

```
        cout << "Car details...\n";
    }
};

class Truck : public Vehicle {
public:
    void display() const override {
        cout << "Truck details...\n";
    }
};
```

4. Abstraction

The program employs **abstraction** through the Vehicle base class. This class provides an interface for common vehicle behaviors (display(), saveToFile(), loadFromFile()), while hiding the implementation details of the derived classes. Additionally, the abstract concept of Vehicle is used to manage a wide variety of vehicle types (e.g., cars, trucks) with shared behaviors but distinct attributes.

Example:

```
class Vehicle {
public:
    virtual void display() const = 0; // Pure virtual function for abstraction
};
```

While the code doesn't use formal abstract classes (with pure virtual methods like calculateSalary() in the example provided), it achieves abstraction by ensuring that users interact with a generalized Vehicle class and its subclasses.

5. File I/O

The program uses file I/O for reading and writing data, as required:

- **Saving data**: The saveToFile() method in Vehicle (and its derived classes Car and Truck) saves vehicle information to a file.
- **Loading data**: The loadFromFile() method reads from a file and reconstructs objects (like Vehicle, Car, Truck) based on the data.

Example:

```
void saveVehiclesToFile(const vector<Vehicle *> &vehicles) {
```

```
    ofstream file(filename);
    if (!file) {
        cerr << "Error: Unable to open file for saving vehicles!" << endl;
        return;
    }

    for (const auto &vehicle : vehicles) {
        vehicle->saveToFile(file);
    }
    file.close();
}
```

6. Exception Handling

The program handles potential errors using **try-catch blocks** to capture exceptions. For example, when loading vehicles from a file, the program catches parsing errors and file I/O issues.

Example:

```
try {
    // Code that may throw an exception
} catch (const exception &e) {
    cerr << "Error: " << e.what() << endl;
}
```

This is implemented in multiple places, such as when reading from or writing to files, ensuring that the program doesn't crash if an error occurs.

7. Menu/Command-line Interface (CLI)

The system provides a **text-based interface** with a menu that allows the user to interact with the system. The menu options include adding vehicles, displaying vehicles, adding customers, saving and loading data, and more.

Example:

```
void showMenu() {
    cout << "=== Transport Logistics System ===" << endl;
    cout << "1. Add Vehicle" << endl;
```

```
    cout << "2. Display Vehicles" << endl;
    cout << "3. Save Vehicles to File" << endl;
    cout << "4. Load Vehicles from File" << endl;
    cout << "5. Add Customer" << endl;
    cout << "6. Display Customers" << endl;
    cout << "7. Save Customers to File" << endl;
    cout << "8. Load Customers from File" << endl;
    cout << "9. Exit" << endl;
    cout << "Enter your choice: ";
}
```

The user can navigate through the options, perform operations like adding a vehicle or displaying customer details, and save/load data to/from files. Each action is handled in a structured way within a loop, making it user-friendly.

Summary

The program meets the functional requirements as follows:

1. **Class Definitions**: 4 classes (Vehicle, Car, Truck, and Customer) are defined, each encapsulating different responsibilities.
2. **Inheritance**: The program uses inheritance with a base class (Vehicle) and derived classes (Car, Truck).
3. **Polymorphism**: Method overriding is demonstrated using virtual functions, such as display().
4. **Abstraction**: The Vehicle class abstracts the details of specific vehicle types, allowing shared behavior across different types.
5. **File I/O**: The program allows saving and loading vehicle and customer data to/from files using fstream.
6. **Exception Handling**: The program uses try-catch blocks to handle potential errors.
7. **Menu/CLI**: A text-based menu system provides an interface for user interaction.

This program implements all of the specified functional requirements, demonstrating key principles of object-oriented programming (OOP) and file management.