SOLID principles in C++

S — Single Responsibility Principle (SRP)

Each class should have only one reason to change.
Focus each class on a single task.

✅ Good Example:

```
class Logger {
public:
    void log(const std::string& msg) {
        std::cout << "LOG: " << msg << std::endl;
    }
};

class FileManager {
public:
    void saveFile(const std::string& filename) {
        std::cout << "Saving file: " << filename << std::endl;
    }
};
```

Here, `Logger` only logs. `FileManager` only handles files. Each has **one job**.

O — Open/Closed Principle (OCP)

Classes should be open for extension, closed for modification.
Add new behavior without changing old code.

✅ Good Example (using polymorphism):

```
class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Draw Circle" << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Draw Rectangle" << std::endl;
    }
};

void renderShape(Shape* shape) {
    shape->draw();
}
```

You can add `Triangle`, `Square`, etc., **without touching** `renderShape`.

L — Liskov Substitution Principle (LSP)

Subclasses must be replaceable for their base classes without breaking functionality.

❌ Bad Example:

```
class Bird {
public:
    virtual void fly() = 0;
};

class Penguin : public Bird {
public:
    void fly() override {
        throw std::runtime_error("Penguins can't fly!");
    }
};
```

Penguin **violates LSP**. It's a bird but can't fly.

✅ Fix:

```
class Bird {
public:
    virtual void eat() = 0;
};

class FlyingBird : public Bird {
public:
    virtual void fly() = 0;
};

class Sparrow : public FlyingBird {
public:
    void eat() override {}
    void fly() override {}
};

class Penguin : public Bird {
public:
    void eat() override {}
};
```

Now, no one forces `Penguin` to fly.

I — Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.

❌ Bad Example:

```
class IMachine {
public:
    virtual void print() = 0;
    virtual void scan() = 0;
    virtual void fax() = 0;
};

class OldPrinter : public IMachine {
public:
    void print() override {}
    void scan() override {} // not used
    void fax() override {}  // not used
};
```

`OldPrinter` is forced to implement unused methods.

✅ Fix:

```
class IPrinter {
public:
    virtual void print() = 0;
};

class IScanner {
public:
    virtual void scan() = 0;
};

class OldPrinter : public IPrinter {
public:
    void print() override {}
};
```

D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

❌ Bad Example:

```
class MySQLDatabase {
public:
    void connect() {}
};

class UserService {
    MySQLDatabase db;
public:
    void login() {
        db.connect();
    }
};
```

`UserService` is tightly coupled with `MySQLDatabase`.

✅ Fix:

```
class IDatabase {
public:
    virtual void connect() = 0;
};

class MySQLDatabase : public IDatabase {
public:
    void connect() override {
        std::cout << "Connected to MySQL\n";
    }
};

class UserService {
    IDatabase* db;
public:
    UserService(IDatabase* database) : db(database) {}
    void login() {
        db->connect();
    }
};
```

Now you can pass `PostgreSQLDatabase`, `MockDatabase`, etc.

✅ Practice Tasks

1. **SRP**
   Create a class that reads a file and logs content. Then split into two classes (Reader, Logger).

2. **OCP**
   Implement `Shape` interface with `Triangle`, `Square`. Call `draw()` without `if` conditions.

3. **LSP**
   Design an `Animal` class. Ensure `Dog` and `Cat` can replace `Animal` without breaking logic.

4. **ISP**
   Design interfaces for `IPlayable`, `ISkippable`. Create classes like `MusicPlayer`,

`VideoPlayer` with only the interfaces they need.

5. **DIP**
   Implement `INotification` with `EmailNotification` and `SMSNotification`. Inject via constructor into `OrderService`.

Here are **15 hands-on practice tasks** (3 for each SOLID principle) using **C++**, sorted by increasing difficulty.

✅ Single Responsibility Principle (SRP)

1. Task Logger Split

- Write a `TaskManager` class that adds, deletes, and logs tasks.
- Then split into:
  - `TaskHandler` for add/delete
  - `Logger` for log

2. Student Report

- Create a `Student` class that stores name, grade, and generates reports.
- Split into:
  - `Student` class
  - `ReportGenerator` class

3. SRP in Bank System

- Class `BankAccount` handles deposits, logs transactions, and sends SMS.

- Refactor into:
    - `TransactionLogger`
    - `SMSNotifier`
    - `BankAccount`

✅ Open/Closed Principle (OCP)

4. Shape Area Calculator

- Write `Shape` base class and `Circle`, `Rectangle`, `Triangle` that implement `area()`.
- Add a new shape without editing old classes.

5. Payment System

- Base class `PaymentMethod`.
- Create derived classes: `CreditCard`, `PayPal`, `Bitcoin`.
- Function `processPayment(PaymentMethod*)`.

6. Sorting Strategy

- Base class `Sorter`
- Derived classes: `BubbleSort`, `QuickSort`, `MergeSort`
- Allow easy extension by adding new sorting algorithm

✅ Liskov Substitution Principle (LSP)

7. Bird Fix

- Create base class `Bird` with `eat()` only.
- `FlyingBird` with `fly()` derived.
- `Penguin`, `Sparrow`, `Eagle` implement correct behaviors.

8. Employee Salary

- Base class `Employee` with `calculateSalary()`.
- `FullTimeEmployee` and `ContractEmployee` return different calculations.
- Should work in any `payroll(Employee*)` function.

9. Shape Draw

- Create `DrawableShape` base class with `draw()`.
- Use it with `Circle`, `Line`, `Rectangle` without breaking `renderAll(vector<DrawableShape*>)`

✅ Interface Segregation Principle (ISP)

10. Media Device

- Create `IMusicPlayer`, `IVideoPlayer`, `IRadio`.
- `Smartphone` implements all.
- `MP3Player` only implements `IMusicPlayer`.

11. Online Editor

- Interfaces: `IPrintable`, `IShareable`, `IEditable`
- `PDFEditor` implements `IPrintable`, `IEditable`
- `DocEditor` implements all

12. Remote Controller

- Interfaces: `IOnOff`, `IVolumeControl`, `IChannelControl`
- `BasicRemote` uses only `IOnOff`
- `SmartRemote` uses all

✅ Dependency Inversion Principle (DIP)

13. Database Injection

- Define interface `IDatabase` with `connect()`
- Implement `MySQL` and `PostgreSQL`
- Inject into `App` class via constructor

14. Message Sender

- Interface `INotificationSender` with `send(msg)`
- Classes: `EmailSender`, `SMSSender`
- Inject into `UserService` class

15. Sensor Logger

- Interface `ISensor`
- `TemperatureSensor`, `PressureSensor` implement it
- `SensorLogger` works with any `ISensor` passed via constructor

Want Guidance?

I can:

- Turn any task into a full project template
- Review your code when you complete them
- Bundle them into a GitHub-ready structure
- Add a simple Makefile or CMakeLists.txt