step 1: OOP & SOLID Principles (Core Foundation)

**Object-Oriented Programming Concepts**

1. **Encapsulation** → wrapping data + methods together (class)  
   👉 Interview Q: *How does C++ achieve encapsulation?*  
   ✅ Answer: By using classes, access specifiers (private, protected, public), getters/setters.
2. **Abstraction** → hiding implementation details, showing only what’s necessary  
   👉 Q: *Difference between abstraction and encapsulation?*  
   ✅ Encapsulation = data hiding inside a class  
   ✅ Abstraction = hiding implementation via abstract classes / interfaces
3. **Inheritance** → one class acquiring properties of another  
   👉 Q: *Types in C++?* (Single, Multiple, Multilevel, Hybrid, Hierarchical)  
   👉 Q: *When to avoid inheritance?* – when “is-a” relation doesn’t hold (prefer composition).
4. **Polymorphism** → many forms
   * Compile-time (function overloading, operator overloading)
   * Run-time (virtual functions, overriding)

👉 Q: *When to use virtual destructor?*  
✅ If base pointer deletes a derived object, destructor must be virtual to avoid memory leaks.

**SOLID Principles**

1. **S – Single Responsibility**: A class should only have one reason to change.  
   *Eg: ParkingLot shouldn’t both manage payments and assign slots.*
2. **O – Open/Closed**: Classes should be open for extension, closed for modification.  
   *Eg: Add new VehicleType without modifying ParkingSlot allocation code → use polymorphism.*
3. **L – Liskov Substitution**: Derived classes must be substitutable for base classes.  
   *Eg: If Car inherits Vehicle, anywhere Vehicle is expected, Car should work correctly.*
4. **I – Interface Segregation**: Clients shouldn’t be forced to depend on unused interfaces.  
   *Eg: Don’t put printReceipt() inside Vehicle interface; only Payment should have it.*
5. **D – Dependency Inversion**: Depend on abstractions, not concrete classes.  
   *Eg: PaymentService should depend on IPaymentMethod (interface), not directly on CreditCard.*

Example With Virtual Destructor

class Base {

public:

virtual ~Base() { // Virtual destructor

cout << "Base Destructor" << endl;

}

};

class Derived : public Base {

public:

~Derived() {

cout << "Derived Destructor" << endl;

}

};

int main() {

Base\* ptr = new Derived();

delete ptr; // Both destructors called ✅

return 0;

}

**🔹 What is the Diamond Problem?**

It happens in **multiple inheritance** when a class inherits from two classes that share a common base.  
This creates **ambiguity** because the derived class may inherit multiple copies of the same base class.

Singleton :

**1. What is the Singleton Pattern?**

✅ Ensures only one instance of a class exists and provides a global access point.  
👉 Example use: Logger, DB Connection Pool, Thread Pool.

#include <iostream>

using namespace std;

class Singleton {

private:

static Singleton\* instance;

Singleton() { cout << "Singleton created\n"; }

public:

static Singleton\* getInstance() {

if (!instance) instance = new Singleton();

return instance;

}

void show() { cout << "Using Singleton\n"; }

};

Singleton\* Singleton::instance = nullptr;

int main() {

Singleton\* s1 = Singleton::getInstance();

Singleton\* s2 = Singleton::getInstance();

cout << (s1 == s2 ? "Same instance\n" : "Different instance\n");

}

**2. What is the Factory Method Pattern?**

✅ Provides an interface for creating objects, but lets subclasses decide which object to create.  
👉 Example use: VehicleFactory creates Car/Bike/Truck.

#include <iostream>

using namespace std;

class Vehicle {

public:

virtual void drive() = 0;

};

class Car : public Vehicle {

public: void drive() { cout << "Driving Car\n"; }

};

class Bike : public Vehicle {

public: void drive() { cout << "Riding Bike\n"; }

};

class VehicleFactory {

public:

static Vehicle\* create(string type) {

if (type == "car") return new Car();

if (type == "bike") return new Bike();

return nullptr;

}

};

int main() {

Vehicle\* v1 = VehicleFactory::create("car");

v1->drive();

Vehicle\* v2 = VehicleFactory::create("bike");

v2->drive();

}

**3. What is the Builder Pattern?**

✅ Used to construct complex objects step by step.  
👉 Example use: Building a Pizza with toppings, crust, size.

#include <iostream>

using namespace std;

class Pizza {

public:

string dough, sauce, topping;

void show() { cout << "Pizza with " << dough << ", " << sauce << ", " << topping << "\n"; }

};

class PizzaBuilder {

protected:

Pizza\* pizza;

public:

PizzaBuilder() { pizza = new Pizza(); }

virtual void buildDough() = 0;

virtual void buildSauce() = 0;

virtual void buildTopping() = 0;

Pizza\* getPizza() { return pizza; }

};

class MargheritaBuilder : public PizzaBuilder {

public:

void buildDough() { pizza->dough = "Thin"; }

void buildSauce() { pizza->sauce = "Tomato"; }

void buildTopping() { pizza->topping = "Cheese"; }

};

class Director {

public:

Pizza\* create(PizzaBuilder& builder) {

builder.buildDough();

builder.buildSauce();

builder.buildTopping();

return builder.getPizza();

}

};

int main() {

MargheritaBuilder builder;

Director director;

Pizza\* p = director.create(builder);

p->show();

}

**🔹 Structural Design Patterns**

**4. What is the Adapter Pattern?**

✅ Converts the interface of a class into another interface clients expect.  
👉 Example use: Integrating a new payment gateway with existing system.

#include <iostream>

using namespace std;

class OldPrinter {

public: void oldPrint() { cout << "Old Printer\n"; }

};

class ModernPrinter {

public: virtual void print() = 0;

};

class Adapter : public ModernPrinter {

OldPrinter\* oldPrinter;

public:

Adapter(OldPrinter\* p) : oldPrinter(p) {}

void print() override { oldPrinter->oldPrint(); }

};

int main() {

OldPrinter old;

ModernPrinter\* printer = new Adapter(&old);

printer->print();

}

**5. What is the Decorator Pattern?**

✅ Dynamically adds behavior to objects without modifying original code.  
👉 Example use: Adding notifications (Email → SMS → Push).

#include <iostream>

using namespace std;

class Notifier {

public: virtual void send() = 0;

};

class EmailNotifier : public Notifier {

public: void send() { cout << "Email sent\n"; }

};

class SMSDecorator : public Notifier {

Notifier\* wrappee;

public:

SMSDecorator(Notifier\* n) : wrappee(n) {}

void send() { wrappee->send(); cout << "SMS sent\n"; }

};

int main() {

Notifier\* n = new SMSDecorator(new EmailNotifier());

n->send();

}

**6. What is the Composite Pattern?**

✅ Treats individual objects and compositions uniformly (tree structures).  
👉 Example use: File System (File + Folder).

#include <iostream>

#include <vector>

using namespace std;

class FileSystem {

public: virtual void show() = 0;

};

class File : public FileSystem {

string name;

public:

File(string n) : name(n) {}

void show() { cout << "File: " << name << "\n"; }

};

class Folder : public FileSystem {

string name;

vector<FileSystem\*> children;

public:

Folder(string n) : name(n) {}

void add(FileSystem\* f) { children.push\_back(f); }

void show() {

cout << "Folder: " << name << "\n";

for (auto c : children) c->show();

}

};

int main() {

Folder root("root");

root.add(new File("a.txt"));

Folder\* sub = new Folder("docs");

sub->add(new File("b.pdf"));

root.add(sub);

root.show();

}

**7. What is the Proxy Pattern?**

✅ Provides a placeholder for another object to control access.  
👉 Example use: Virtual Proxy for expensive objects (like large images).

#include <iostream>

using namespace std;

class Image {

public: virtual void display() = 0;

};

class RealImage : public Image {

string filename;

public:

RealImage(string f) : filename(f) { cout << "Loading " << f << "\n"; }

void display() { cout << "Displaying " << filename << "\n"; }

};

class ProxyImage : public Image {

string filename;

RealImage\* realImg = nullptr;

public:

ProxyImage(string f) : filename(f) {}

void display() {

if (!realImg) realImg = new RealImage(filename);

realImg->display();

}

};

int main() {

Image\* img = new ProxyImage("photo.png");

img->display(); // loads + displays

img->display(); // reuses cached image

}

**🔹 Behavioral Design Patterns**

**8. What is the Strategy Pattern?**

✅ Defines a family of algorithms, encapsulates each one, and makes them interchangeable.  
👉 Example use: Payment strategy (CreditCard, UPI, Wallet).

#include <iostream>

using namespace std;

class Payment {

public: virtual void pay(int amount) = 0;

};

class CreditCard : public Payment {

public: void pay(int amount) { cout << "Paid " << amount << " with Credit Card\n"; }

};

class UPI : public Payment {

public: void pay(int amount) { cout << "Paid " << amount << " with UPI\n"; }

};

class Order {

Payment\* method;

public:

Order(Payment\* p) : method(p) {}

void checkout(int amt) { method->pay(amt); }

};

int main() {

Order o1(new CreditCard());

o1.checkout(100);

Order o2(new UPI());

o2.checkout(200);

}

**9. What is the Observer Pattern?**

✅ One-to-many dependency: when subject changes, observers get notified.  
👉 Example use: Notification system (user gets SMS, Email).

#include <iostream>

#include <vector>

using namespace std;

class Observer {

public: virtual void update(string msg) = 0;

};

class User : public Observer {

string name;

public:

User(string n) : name(n) {}

void update(string msg) { cout << name << " received: " << msg << "\n"; }

};

class NotificationService {

vector<Observer\*> users;

public:

void subscribe(Observer\* u) { users.push\_back(u); }

void notify(string msg) { for (auto u : users) u->update(msg); }

};

int main() {

User u1("Alice"), u2("Bob");

NotificationService service;

service.subscribe(&u1);

service.subscribe(&u2);

service.notify("New Offer!");

}

**10. What is the Command Pattern?**

✅ Encapsulates a request as an object, allowing undo/redo operations.  
👉 Example use: Text editor (undo/redo).

#include <iostream>

#include <vector>

using namespace std;

class Command {

public: virtual void execute() = 0;

};

class Light {

public: void on() { cout << "Light ON\n"; }

void off() { cout << "Light OFF\n"; }

};

class LightOn : public Command {

Light\* light;

public: LightOn(Light\* l) : light(l) {}

void execute() { light->on(); }

};

class Remote {

vector<Command\*> history;

public:

void press(Command\* c) { c->execute(); history.push\_back(c); }

};

int main() {

Light light;

Remote remote;

Command\* on = new LightOn(&light);

remote.press(on);

}

**11. What is the State Pattern?**

✅ Allows an object to change its behavior when its state changes.  
👉 Example use: Vending machine (Idle, HasMoney, Dispensing).

#include <iostream>

using namespace std;

class State {

public: virtual void handle() = 0;

};

class Idle : public State {

public: void handle() { cout << "Machine is Idle\n"; }

};

class Working : public State {

public: void handle() { cout << "Machine is Working\n"; }

};

class Machine {

State\* state;

public:

Machine(State\* s) : state(s) {}

void setState(State\* s) { state = s; }

void request() { state->handle(); }

};

int main() {

Machine m(new Idle());

m.request();

m.setState(new Working());

m.request();

}

**12. What is the Chain of Responsibility Pattern?**

✅ Passes request along a chain until one handler processes it.  
👉 Example use: Logging levels (Info → Debug → Error).

#include <iostream>

using namespace std;

class Handler {

protected: Handler\* next = nullptr;

public:

void setNext(Handler\* n) { next = n; }

virtual void handle(int level) {

if (next) next->handle(level);

}

};

class InfoHandler : public Handler {

public: void handle(int level) {

if (level == 1) cout << "Info log\n";

else Handler::handle(level);

} };

class ErrorHandler : public Handler {

public: void handle(int level) {

if (level == 2) cout << "Error log\n";

else Handler::handle(level);

} };

int main() {

InfoHandler info; ErrorHandler error;

info.setNext(&error);

info.handle(1);

info.handle(2);

}

**🔹 Concurrency Patterns**

**13. Producer-Consumer Problem**

✅ One or more producers generate data, consumers use it.  
👉 Solved using mutex + condition\_variable in C++.

**14. Thread Pool Pattern**

✅ Reuses a fixed set of worker threads to execute tasks.  
👉 Used in web servers for handling requests.

**🔹 1. Parking Lot System**

**Q: How would you design a Parking Lot System?**  
✅ **Answer outline**:

* Entities: ParkingLot, ParkingFloor, ParkingSpot, Vehicle (Car/Bike/Truck), Ticket, Payment.
* Key requirements: assign nearest free spot, generate ticket, calculate fees, support multiple vehicle types.
* Patterns:
  + Singleton → ParkingLot (only one lot manager).
  + Factory → Create Vehicle objects.
  + Strategy → Different pricing algorithms (weekday/weekend).

**🔹 2. Elevator System**

**Q: Design an Elevator system.**  
✅ **Answer outline**:

* Entities: ElevatorController, Elevator, Floor, Request.
* Key requirements: handle multiple elevators, schedule efficiently (nearest-car algorithm).
* Patterns:
  + Observer → Notify floors when elevator arrives.
  + State → Elevator states (Idle, Moving, DoorOpen).
  + Strategy → Different scheduling policies.

**🔹 3. Food Delivery (Zomato/Swiggy)**

**Q: How to design a food delivery app?**  
✅ **Answer outline**:

* Entities: Customer, Restaurant, Menu, Order, DeliveryPartner, Payment.
* Key requirements: browse restaurants, place order, assign delivery partner.
* Patterns:
  + Observer → Notify customer about order status.
  + Strategy → Payment options (UPI, Card, Wallet).
  + Factory → Restaurant can create MenuItems.

**🔹 4. BookMyShow / Ticket Booking**

**Q: Design a ticket booking system.**  
✅ **Answer outline**:

* Entities: Theater, Screen, Show, Seat, Booking, Payment.
* Key requirements: check seat availability, hold seat until payment, prevent double-booking.
* Patterns:
  + Singleton → Booking manager.
  + Strategy → Payment methods.
  + Observer → Notify user when booking confirmed.

**🔹 5. Ride Sharing (Uber/Ola)**

**Q: How would you design Uber?**  
✅ **Answer outline**:

* Entities: Rider, Driver, Trip, Location, Payment.
* Key requirements: match rider with nearby driver, calculate fare, tracking.
* Patterns:
  + Strategy → Fare calculation.
  + Observer → Notify rider/driver on trip updates.
  + Singleton → TripManager.

**🔹 6. ATM / Vending Machine**

**Q: Design an ATM.**  
✅ **Answer outline**:

* Entities: ATM, Card, Account, Transaction, CashDispenser.
* Key requirements: withdraw, deposit, check balance, pin validation.
* Patterns:
  + State → ATM states (Idle, CardInserted, TransactionInProgress).
  + Strategy → Different withdrawal limits per account type.

**🔹 7. Notification Service**

**Q: How to design a Notification Service?**  
✅ **Answer outline**:

* Entities: Notification, Channel (SMS/Email/Push), User.
* Key requirements: send notification via multiple channels, scalable.
* Patterns:
  + Observer → Users subscribe to notifications.
  + Strategy → Choose channel dynamically.
  + Decorator → Add new notification types without modifying core code.

**🔹 8. Rate Limiter**

**Q: How would you design a Rate Limiter?**  
✅ **Answer outline**:

* Entities: RateLimiter, UserRequest.
* Algorithms: Token Bucket, Leaky Bucket, Fixed Window, Sliding Window.
* Key requirements: prevent API abuse, different rules for different users.
* Patterns:
  + Singleton → Global limiter instance.
  + Strategy → Different rate limiting algorithms.

**🔹 9. Cache (LRU, LFU)**

**Q: Design an LRU Cache.**  
✅ **Answer outline**:

* Entities: Cache, Node, DoublyLinkedList.
* Key requirements: O(1) get() and put().
* Approach: HashMap + Doubly Linked List.
* Patterns:
  + Singleton → Global cache.
  + Strategy → Replacement policies (LRU, LFU).

**🔹 10. Messaging Queue (Kafka-like)**

**Q: Design a Messaging Queue.**  
✅ **Answer outline**:

* Entities: Producer, Consumer, MessageQueue.
* Key requirements: publish-subscribe model, ordered delivery, persistence.
* Patterns:
  + Observer → Consumers subscribe to topics.
  + Singleton → Queue manager.
  + Producer-Consumer → Thread synchronization.