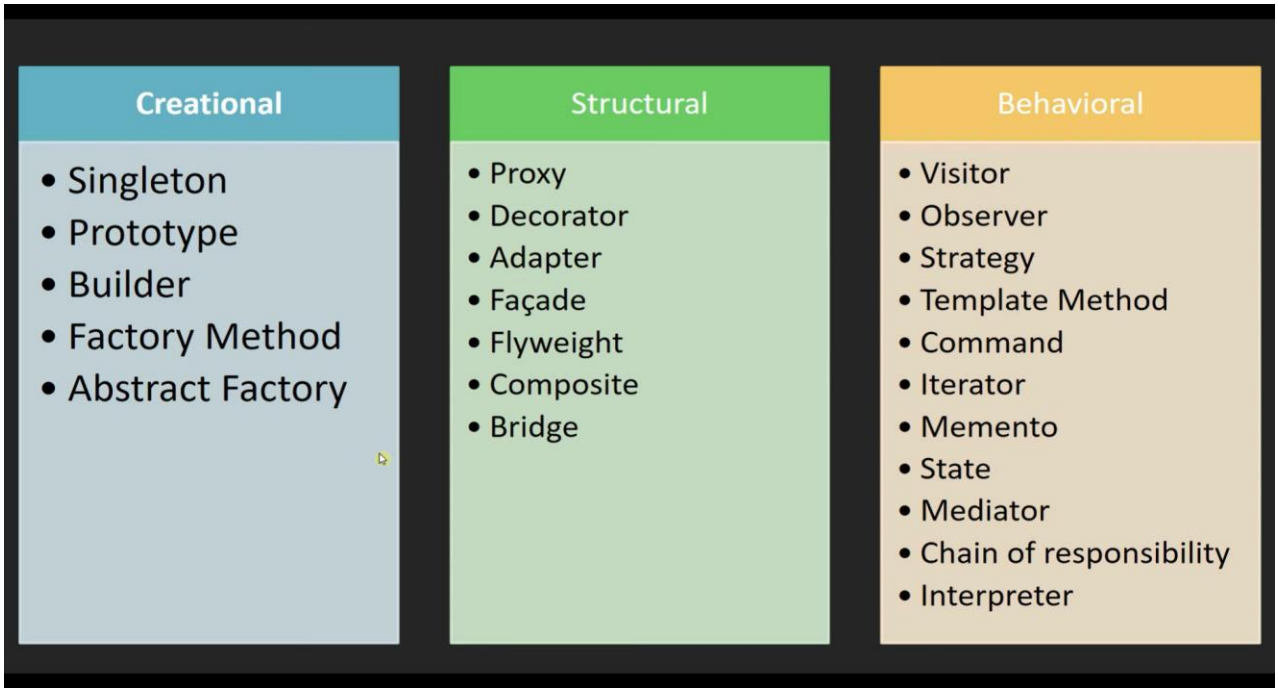
Design Pattern in C++

Types :

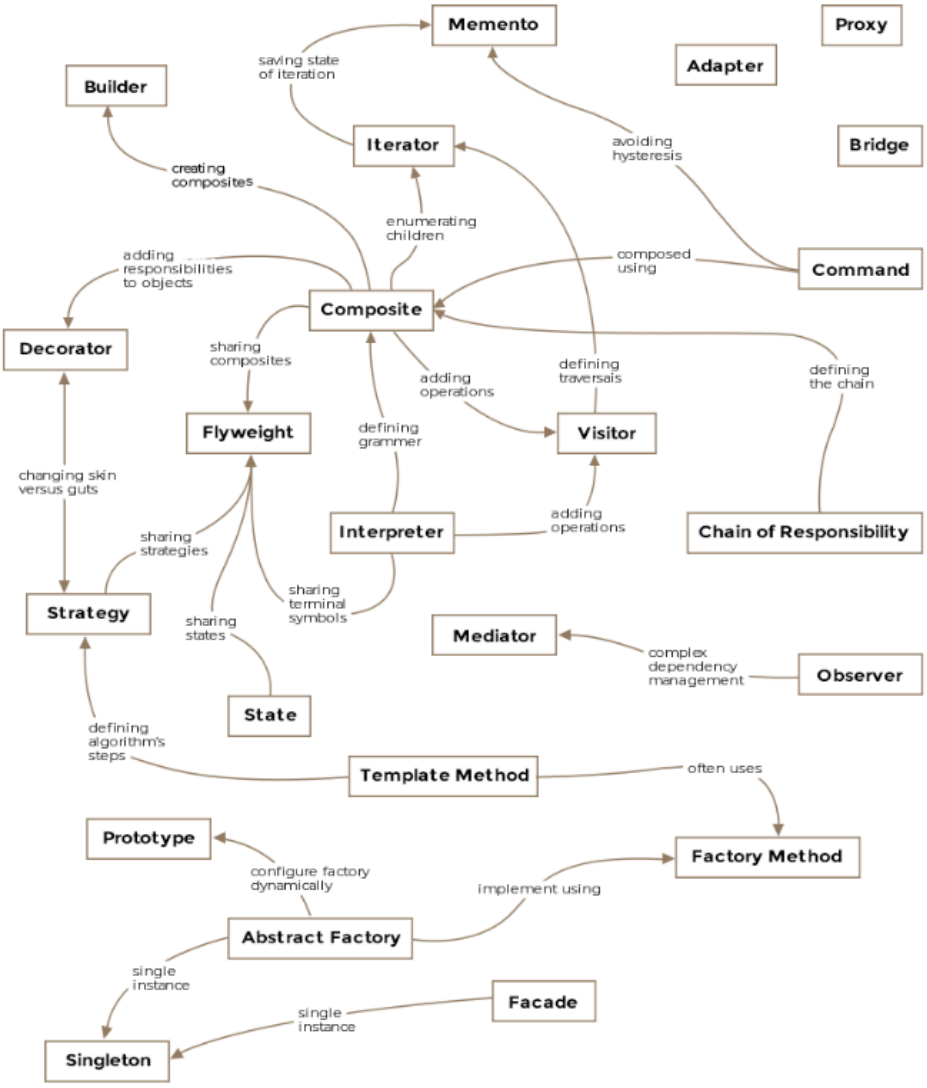
1. **Creational**
2. **Structural**
3. **Behavioral**



Creational Design Pattern:
As the name suggests, it provides the object or classes creation mechanism that enhance the flexibilities and reusability of the existing code. They reduce the dependency and controlling how the use interaction with our class so we wouldn't deal with the complex construction. Below are the various design pattern of creational design pattern.

Structural Design Patterns:
Structural Design Patterns mainly responsible for assemble object and classes into a larger structure making sure that these structure should be flexible and efficient. They are very essential for enhancing readability and maintainability of the code. It also ensure that functionalities are properly separated, encapsulated. It reduces the minimal interface between interdependent things.

Behavioral Design Patterns:



Design Pattern Relationships

Creational Patterns: **Singleton Pattern**

Ensures that a class has only one instance and provides a global point of access to it.

```
#include <iostream>

class Singleton {
private:
    static Singleton* instance; // Static instance
    Singleton() {} // Private constructor to prevent instantiation

public:
    static Singleton* getInstance() {
```

```
        if (!instance) {
            instance = new Singleton();
        }
        return instance;
    }

    void showMessage() {
        std::cout << "Singleton Instance\n";
    }
};

// Initialize static member
Singleton* Singleton::instance = nullptr;

int main() {
    Singleton* obj = Singleton::getInstance();
    obj->showMessage();
    return 0;
}
```

Creational Patterns: *Factory Method Pattern*

Provides an interface for creating objects but lets subclasses alter the type of objects that will be created.

```
#include <iostream>

class Product {
public:
    virtual void use() = 0; // Abstract method
};

class ConcreteProductA : public Product {
public:
    void use() override {
        std::cout << "Using ConcreteProductA\n";
    }
};

class ConcreteProductB : public Product {
public:
    void use() override {
        std::cout << "Using ConcreteProductB\n";
    }
};

class Factory {
public:
    static Product* createProduct(char type) {
```

```
        if (type == 'A') return new ConcreteProductA();
        if (type == 'B') return new ConcreteProductB();
        return nullptr;
    }
};

int main() {
    Product* product = Factory::createProduct('A');
    product->use();
    delete product;
    return 0;
}
```

Creational Patterns: **Abstract Factory Pattern**

Provides an interface for creating families of related objects without specifying their concrete classes.

```
#include <iostream>

class AbstractProductA {

public:

    virtual void use() = 0;

};

class AbstractProductB {

public:

    virtual void play() = 0;

};

// Concrete products

class ProductA1 : public AbstractProductA {

public:

    void use() override {

        std::cout << "Using ProductA1\n";

    }

};

class ProductB1 : public AbstractProductB {
```

```
public:
    void play() override {
        std::cout << "Playing with ProductB1\n";
    }
};
// Abstract Factory
class AbstractFactory {
public:
    virtual AbstractProductA* createProductA() = 0;
    virtual AbstractProductB* createProductB() = 0;
};
// Concrete Factory
class ConcreteFactory1 : public AbstractFactory {
public:
    AbstractProductA* createProductA() override {
        return new ProductA1();
    }
    AbstractProductB* createProductB() override {
        return new ProductB1();
    }
};
int main() {
    AbstractFactory* factory = new ConcreteFactory1();
    AbstractProductA* productA = factory->createProductA();
    AbstractProductB* productB = factory->createProductB();
    productA->use();
    productB->play();
```

```
    delete productA;
    delete productB;
    delete factory;
    return 0;
}
```

Creational Patterns: ***Builder Pattern***

Separates the construction of a complex object from its representation.

```
#include <iostream>
#include <string>
class Product {
public:
    std::string partA;
    std::string partB;
    void show() {
        std::cout << "Product with " << partA << " and " << partB << "\n";
    }
};
// Builder Interface
class Builder {
public:
    virtual void buildPartA() = 0;
    virtual void buildPartB() = 0;
    virtual Product* getResult() = 0;
};
// Concrete Builder
class ConcreteBuilder : public Builder {
private:
```

```
    Product* product;
public:
    ConcreteBuilder() { product = new Product(); }
    void buildPartA() override { product->partA = "Part A"; }
    void buildPartB() override { product->partB = "Part B"; }
    Product* getResult() override { return product; }
};
// Director
class Director {
public:
    void construct(Builder* builder) {
        builder->buildPartA();
        builder->buildPartB();
    }
};
int main() {
    ConcreteBuilder builder;
    Director director;
    director.construct(&builder);
    Product* product = builder.getResult();
    product->show();
    delete product;
    return 0;
}
```

Creational Patterns: **Prototype Pattern**

Creates new objects by copying an existing object.

```
#include <iostream>
```

```
class Prototype {
public:
    virtual Prototype* clone() = 0; // Clone method
    virtual void show() = 0;
    virtual ~Prototype() = default;
};
class ConcretePrototype : public Prototype {
public:
    int value;
    ConcretePrototype(int v) : value(v) {}
    Prototype* clone() override {
        return new ConcretePrototype(*this);
    }
    void show() override {
        std::cout << "ConcretePrototype with value " << value << "\n";
    }
};
int main() {
    ConcretePrototype* original = new ConcretePrototype(10);
    ConcretePrototype* copy = static_cast<ConcretePrototype*>(original->clone());
    original->show();
    copy->show();
    delete original;
    delete copy;
    return 0;
}
```

Structural Design Patterns in C++

Structural design patterns focus on organizing classes and objects to form larger structures while keeping systems flexible and efficient. These patterns simplify relationships between entities.

Structural Design Patterns : Adapter Pattern

Converts one interface into another that a client expects.

```
#include <iostream>

// Target Interface

class Target {

public:

    virtual void request() {

        std::cout << "Target: Default request implementation\n";

    }

};

// Adaptee (Existing class with different interface)

class Adaptee {

public:

    void specificRequest() {

        std::cout << "Adaptee: Specific request implementation\n";

    }

};

// Adapter (Bridges Target and Adaptee)

class Adapter : public Target {

private:

    Adaptee* adaptee;

public:

    Adapter(Adaptee* a) : adaptee(a) {}
```

```
    void request() override {
        adaptee->specificRequest();
    }
};
int main() {
    Adaptee* adaptee = new Adaptee();
    Target* adapter = new Adapter(adaptee);
    adapter->request();
    delete adaptee;
    delete adapter;
    return 0;
}
```

Use case: When you need to work with an existing class but its interface is incompatible.

Structural Design Patterns : Bridge Pattern

Decouples abstraction from its implementation, allowing them to vary independently.

```
#include <iostream>
// Implementor Interface
class Implementor {
public:
    virtual void operationImpl() = 0;
};
// Concrete Implementations
class ConcreteImplementorA : public Implementor {
public:
    void operationImpl() override {
        std::cout << "ConcreteImplementorA operation\n";
    }
```

```
};
class ConcreteImplementorB : public Implementor {
public:
    void operationImpl() override {
        std::cout << "ConcreteImplementorB operation\n";
    }
};
// Abstraction
class Abstraction {
protected:
    Implementor* implementor;
public:
    Abstraction(Implementor* impl) : implementor(impl) {}
    virtual void operation() {
        implementor->operationImpl();
    }
};
int main() {
    Implementor* impl = new ConcreteImplementorA();
    Abstraction* abs = new Abstraction(impl);
    abs->operation();
    delete impl;
    delete abs;
    return 0;
}
```

Use case: When you need to separate abstraction from implementation for flexibility.

Structural Design Patterns : Composite Pattern

Treats individual objects and compositions of objects uniformly.

```
#include <iostream>
#include <vector>
// Component Interface
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() = default;
};
// Leaf Class
class Leaf : public Component {
public:
    void operation() override {
        std::cout << "Leaf operation\n";
    }
};
// Composite Class
class Composite : public Component {
private:
    std::vector<Component*> children;
public:
    void add(Component* component) {
        children.push_back(component);
    }
```

```
    void operation() override {
        std::cout << "Composite operation\n";
        for (Component* child : children) {
            child->operation();
        }
    }
};
int main() {
    Composite* composite = new Composite();
    composite->add(new Leaf());
    composite->add(new Leaf());
    composite->operation();
    delete composite;
    return 0;
}
```

Use case: When objects need to be treated as part of a hierarchy or tree structure.

Structural Design Patterns : Decorator Pattern

Adds functionality to objects dynamically.

```
#include <iostream>
// Base Component
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() = default;
};
// Concrete Component
class ConcreteComponent : public Component {
```

```
public:
    void operation() override {
        std::cout << "ConcreteComponent operation\n";
    }
};
// Decorator
class Decorator : public Component {
protected:
    Component* component;
public:
    Decorator(Component* comp) : component(comp) {}
    void operation() override {
        component->operation();
    }
};
// Concrete Decorator
class ConcreteDecorator : public Decorator {
public:
    ConcreteDecorator(Component* comp) : Decorator(comp) {}
    void operation() override {
        Decorator::operation();
        std::cout << "ConcreteDecorator additional operation\n";
    }
};

int main() {
    Component* component = new ConcreteComponent();
```

```
    Component* decorated = new ConcreteDecorator(component);
    decorated->operation();
    delete component;
    delete decorated;
    return 0;
}
```

Use case: When you need to add behavior without modifying existing classes.

Structural Design Patterns : Facade Pattern

Provides a simplified interface to a complex subsystem.

```
#include <iostream>
// Subsystem classes
class SubsystemA {
public:
    void operationA() { std::cout << "SubsystemA operation\n"; }
};
class SubsystemB {
public:
    void operationB() { std::cout << "SubsystemB operation\n"; }
};
// Facade
class Facade {
private:
    SubsystemA* a;
    SubsystemB* b;
public:
    Facade() {
```

```
        a = new SubsystemA();
        b = new SubsystemB();
    }
    void operation() {
        a->operationA();
        b->operationB();
    }
    ~Facade() {
        delete a;
        delete b;
    }
};
int main() {
    Facade* facade = new Facade();
    facade->operation();
    delete facade;
    return 0;
}
```

Use case: When simplifying a complex system's interface.

Structural Design Patterns : Flyweight Pattern

Minimizes memory use by sharing objects.

```
#include <iostream>
#include <unordered_map>
// Flyweight Interface
class Flyweight {
public:
    virtual void operation() = 0;
```

```
};
// Concrete Flyweight
class ConcreteFlyweight : public Flyweight {
public:
    void operation() override {
        std::cout << "ConcreteFlyweight operation\n";
    }
};
// Flyweight Factory
class FlyweightFactory {
private:
    std::unordered_map<std::string, Flyweight*> flyweights;
public:
    Flyweight* getFlyweight(const std::string& key) {
        if (flyweights.find(key) == flyweights.end()) {
            flyweights[key] = new ConcreteFlyweight();
        }
        return flyweights[key];
    }
    ~FlyweightFactory() {
        for (auto& pair : flyweights)
            delete pair.second;
    }
};
int main() {
    FlyweightFactory factory;
    Flyweight* fw1 = factory.getFlyweight("A");
```

```
    Flyweight* fw2 = factory.getFlyweight("A");
    fw1->operation();
    fw2->operation();
    return 0;
}
```

Use case: When large numbers of similar objects are used to save memory.

Structural Design Patterns : Proxy Pattern

Provides a placeholder for another object to control access.

```
#include <iostream>
// Real Subject
class RealSubject {
public:
    void request() {
        std::cout << "RealSubject request\n";
    }
};
// Proxy
class Proxy {
private:
    RealSubject* realSubject;
public:
    Proxy() : realSubject(nullptr) {}
    void request() {
        if (!realSubject) {
            realSubject = new RealSubject();
        }
        realSubject->request();
```

```
    }
    ~Proxy() {
        delete realSubject;
    }
};
int main() {
    Proxy proxy;
    proxy.request();
    proxy.request();
    return 0;
}
```

Use case: When you need controlled access to a resource.

Behavioral pattern in c++

Behavioral design patterns focus on communication between objects, defining how they interact while maintaining flexibility and reducing dependencies.

Behavioral pattern : Chain of Responsibility Pattern

Passes a request along a chain of handlers.

```
#include <iostream>
// Abstract Handler
class Handler {
protected:
    Handler* next;
public:
    Handler() : next(nullptr) {}
    void setNext(Handler* handler) { next = handler; }
    virtual void handleRequest(int level) {
        if (next) next->handleRequest(level);
```

```
    }
};
// Concrete Handlers
class ConcreteHandlerA : public Handler {
public:
    void handleRequest(int level) override {
        if (level == 1) {
            std::cout << "Handled by ConcreteHandlerA\n";
        } else if (next) {
            next->handleRequest(level);
        }
    }
};
class ConcreteHandlerB : public Handler {
public:
    void handleRequest(int level) override {
        if (level == 2) {
            std::cout << "Handled by ConcreteHandlerB\n";
        } else if (next) {
            next->handleRequest(level);
        }
    }
};
int main() {
    ConcreteHandlerA handlerA;
    ConcreteHandlerB handlerB;
    handlerA.setNext(&handlerB);
```

```
    handlerA.handleRequest(1);
    handlerA.handleRequest(2);
    return 0;
}
```

Use case: When multiple objects should handle a request dynamically.

Behavioral pattern : Command Pattern

Encapsulates a request as an object.

```
#include <iostream>
// Command Interface
class Command {
public:
    virtual void execute() = 0;
};
// Receiver
class Receiver {
public:
    void action() { std::cout << "Receiver action executed\n"; }
};
// Concrete Command
class ConcreteCommand : public Command {
private:
    Receiver* receiver;
public:
    ConcreteCommand(Receiver* r) : receiver(r) {}
    void execute() override { receiver->action(); }
```

```
};

// Invoker

class Invoker {

private:

    Command* command;

public:

    void setCommand(Command* cmd) { command = cmd; }

    void executeCommand() { command->execute(); }

};

int main() {

    Receiver receiver;

    ConcreteCommand command(&receiver);

    Invoker invoker;

    invoker.setCommand(&command);

    invoker.executeCommand();

    return 0;

}
```

Use case: When operations need to be encapsulated and executed later.

Behavioral pattern : Iterator Pattern

Provides a way to access elements of a collection sequentially without exposing its underlying representation.

```
#include <iostream>

#include <vector>

// Iterator Interface

class Iterator {

public:
```

```
    virtual bool hasNext() = 0;
    virtual int next() = 0;
};
// Concrete Iterator
class ConcreteIterator : public Iterator {
private:
    std::vector<int> data;
    size_t index;
public:
    ConcreteIterator(std::vector<int> d) : data(d), index(0) {}
    bool hasNext() override {
        return index < data.size();
    }
    int next() override {
        return data[index++];
    }
};
int main() {
    std::vector<int> numbers = {1, 2, 3, 4};
    ConcreteIterator iterator(numbers);
    while (iterator.hasNext()) {
        std::cout << iterator.next() << " ";
    }
    return 0;
}
```

Use case: When sequential access to a collection is needed without exposing details.

Behavioral pattern : Mediator Pattern

Centralizes communication between objects.

```
#include <iostream>
#include <vector>
// Forward declaration
class Colleague;

// Mediator Interface
class Mediator {
public:
    virtual void sendMessage(const std::string& message, Colleague* sender) = 0;
};
// Colleague Interface
class Colleague {
protected:
    Mediator* mediator;
public:
    Colleague(Mediator* m) : mediator(m) {}
    virtual void receive(const std::string& message) = 0;
};
// Concrete Colleague
class ConcreteColleague : public Colleague {
public:
    ConcreteColleague(Mediator* m) : Colleague(m) {}
    void send(const std::string& message) {
        mediator->sendMessage(message, this);
    }
```

```
    void receive(const std::string& message) override {
        std::cout << "Received message: " << message << "\n";
    }
};
// Concrete Mediator
class ConcreteMediator : public Mediator {
private:
    std::vector<Colleague*> colleagues;
public:
    void addColleague(Colleague* colleague) {
        colleagues.push_back(colleague);
    }
    void sendMessage(const std::string& message, Colleague* sender) override {
        for (Colleague* colleague : colleagues) {
            if (colleague != sender) {
                colleague->receive(message);
            }
        }
    }
};
int main() {
    ConcreteMediator mediator;
    ConcreteColleague colleague1(&mediator);
    ConcreteColleague colleague2(&mediator);
    mediator.addColleague(&colleague1);
    mediator.addColleague(&colleague2);
    colleague1.send("Hello, World!");
```

```
    return 0;
}
```

Use case: When direct communication between objects needs to be minimized.

Observer Pattern

Defines a dependency between objects so that when one changes state, all dependents are notified.

```
#include <iostream>
#include <vector>
// Observer Interface
class Observer {
public:
    virtual void update(int state) = 0;
};
// Subject
class Subject {
private:
    std::vector<Observer*> observers;
    int state;
public:
    void attach(Observer* obs) { observers.push_back(obs); }
    void setState(int s) {
        state = s;
        notify();
    }
    void notify() {
```

```
        for (Observer* obs : observers) {
            obs->update(state);
        }
    }
};
// Concrete Observer
class ConcreteObserver : public Observer {
public:
    void update(int state) override {
        std::cout << "Observer updated with state: " << state << "\n";
    }
};
int main() {
    Subject subject;
    ConcreteObserver observer1, observer2;
    subject.attach(&observer1);
    subject.attach(&observer2);
    subject.setState(10);
    return 0;
}
```

Use case: When changes in one object should trigger updates in others.

State Pattern

Allows an object to change its behavior when its internal state changes.

```
#include <iostream>
// Forward declaration
class State;
// Context
```

```
class Context {
private:
    State* state;
public:
    Context(State* initialState);
    void setState(State* newState);
    void request();
};
// Abstract State
class State {
public:
    virtual void handle(Context* context) = 0;
};
// Concrete States
class ConcreteStateA : public State {
public:
    void handle(Context* context) override;
};
class ConcreteStateB : public State {
public:
    void handle(Context* context) override;
};
// Context implementation
Context::Context(State* initialState) : state(initialState) {}
void Context::setState(State* newState) {
    state = newState;
}
```

```
void Context::request() {
    state->handle(this);
}
// State transitions
void ConcreteStateA::handle(Context* context) {
    std::cout << "State A handling request, switching to State B\n";
    context->setState(new ConcreteStateB());
}
void ConcreteStateB::handle(Context* context) {
    std::cout << "State B handling request, switching to State A\n";
    context->setState(new ConcreteStateA());
}
int main() {
    Context context(new ConcreteStateA());
    context.request();
    context.request();
    context.request();
    return 0;
}
```

Use case: When an object needs to change behavior dynamically based on internal state.

Strategy Pattern

Defines a family of algorithms and makes them interchangeable.

```
#include <iostream>
// Strategy Interface
class Strategy {
public:
    virtual void execute() = 0;
```

```
};
// Concrete Strategies
class ConcreteStrategyA : public Strategy {
public:
    void execute() override {
        std::cout << "Executing Strategy A\n";
    }
};
class ConcreteStrategyB : public Strategy {
public:
    void execute() override {
        std::cout << "Executing Strategy B\n";
    }
};
// Context using strategy
class Context {
private:
    Strategy* strategy;
public:
    Context(Strategy* strat) : strategy(strat) {}
    void setStrategy(Strategy* strat) {
        strategy = strat;
    }
    void executeStrategy() {
        strategy->execute();
    }
};
```

```
int main() {
    Context context(new ConcreteStrategyA());
    context.executeStrategy();
    context.setStrategy(new ConcreteStrategyB());
    context.executeStrategy();
    return 0;
}
```

Use case: When multiple algorithms are available, and selection happens at runtime.

Template Method Pattern

Defines the structure of an algorithm but lets subclasses implement specific steps.

```
#include <iostream>
// Abstract Class with Template Method
class AbstractClass {
public:
    void templateMethod() {
        step1();
        step2();
        step3();
    }
    virtual void step1() {
        std::cout << "Default Step 1\n";
    }
    virtual void step2() = 0; // Must be implemented by subclasses
    virtual void step3() {
        std::cout << "Default Step 3\n";
    }
```

```
};

// Concrete Implementations

class ConcreteClassA : public AbstractClass {

public:

    void step2() override {

        std::cout << "ConcreteClassA Step 2\n";

    }

};

class ConcreteClassB : public AbstractClass {

public:

    void step2() override {

        std::cout << "ConcreteClassB Step 2\n";

    }

};

int main() {

    ConcreteClassA objA;

    ConcreteClassB objB;

    std::cout << "Executing ConcreteClassA:\n";

    objA.templateMethod();

    std::cout << "\nExecuting ConcreteClassB:\n";

    objB.templateMethod();

    return 0;

}
```

Use case: When an algorithm's structure is fixed, but steps need customization.

Visitor Pattern

Allows adding new operations to a class hierarchy without modifying existing classes.

```
#include <iostream>
#include <vector>
class ConcreteElementA;
class ConcreteElementB;

// Visitor Interface
class Visitor {
public:
    virtual void visit(ConcreteElementA* element) = 0;
    virtual void visit(ConcreteElementB* element) = 0;
};
// Element Interface
class Element {
public:
    virtual void accept(Visitor* visitor) = 0;
};
// Concrete Elements
class ConcreteElementA : public Element {
public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
    void operationA() {
        std::cout << "Operation A\n";
    }
};
class ConcreteElementB : public Element {
```

```
public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
    void operationB() {
        std::cout << "Operation B\n";
    }
};
// Concrete Visitor
class ConcreteVisitor : public Visitor {
public:
    void visit(ConcreteElementA* element) override {
        std::cout << "Visiting ConcreteElementA\n";
        element->operationA();
    }
    void visit(ConcreteElementB* element) override {
        std::cout << "Visiting ConcreteElementB\n";
        element->operationB();
    }
};
int main() {
    std::vector<Element*> elements = { new ConcreteElementA(), new ConcreteElementB() };
    ConcreteVisitor visitor;
    for (Element* elem : elements) {
        elem->accept(&visitor);
    }
```

```
    for (Element* elem : elements) {
        delete elem;
    }
    return 0;
}
```

Use case: When adding new behaviors to a class hierarchy without modifying existing classes.

Polymorphism : it means **"many forms"** and allows the same function or operator to work in different ways based on object and data.

C++ supports **two types of polymorphism**:

1. **Compile-time (Static) Polymorphism**
    - Function Overloading
    - Operator Overloading
    - Templates
2. **Run-time (Dynamic) Polymorphism**
    - Virtual Functions (Achieved using inheritance and pointers)

1. Compile-Time Polymorphism (Static Binding)

(A) Function Overloading

A class can have multiple functions with the same name but **different parameters**.

```
#include <iostream>
class Example {
public:
    void show() {
        std::cout << "No arguments\n";
    }
    void show(int x) {
        std::cout << "Integer argument: " << x << "\n";
    }
```

```
    void show(double x) {
        std::cout << "Double argument: " << x << "\n";
    }
};
int main() {
    Example obj;
    obj.show();
    obj.show(10);
    obj.show(3.14);
    return 0;
}
```

(B) Operator Overloading

You can redefine how operators work for **user-defined types**.

```
#include <iostream>
class Complex {
public:
    int real, imag;
    Complex(int r, int i) : real(r), imag(i) {}
    // Overloading the `+` operator
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }
    void display() {
        std::cout << real << " + " << imag << "i\n";
    }
};
```

```
int main() {
    Complex c1(3, 4), c2(2, 5);
    Complex c3 = c1 + c2; // Calls operator+ function
    c3.display();
    return 0;
}
```

(C) Templates (Function and Class)

Templates allow **compile-time** polymorphism by working with **different data types**.

```
#include <iostream>
// Function Template
template <typename T>
T add(T a, T b) {
    return a + b;
}
int main() {
    std::cout << "Sum: " << add(10, 20) << "\n";       // int
    std::cout << "Sum: " << add(5.5, 2.5) << "\n";     // double
    return 0;
}
```

2. Run-Time Polymorphism (Dynamic Binding)

Run-time polymorphism is achieved through **inheritance** and **virtual functions**. It allows a **base class pointer** to call **derived class functions dynamically**.

(A) Virtual Functions (Method Overriding)

A **virtual function** in the base class allows calling the **overridden function in derived class**.

```
#include <iostream>

class Base {

public:

    virtual void show() { // Virtual function

        std::cout << "Base class function\n";

    }

};

class Derived : public Base {

public:

    void show() override { // Overriding the base class function

        std::cout << "Derived class function\n";

    }

};

int main() {

    Base* bptr;  // Base class pointer

    Derived d;

    bptr = &d;   // Base class pointer points to Derived object

    bptr->show(); // Calls Derived's show() function (runtime polymorphism)

    return 0;

}
```

(B) Pure Virtual Functions and Abstract Classes

A **pure virtual function** (`= 0`) makes a class **abstract**, meaning **you cannot instantiate it directly**.

```
#include <iostream>

class Shape {

public:

    virtual void draw() = 0; // Pure virtual function (must be overridden)

};

class Circle : public Shape {

public:

    void draw() override {

        std::cout << "Drawing a Circle\n";

    }

};

int main() {

    // Shape s;  // ❌ Error: Cannot instantiate abstract class

    Shape* shape = new Circle();

    shape->draw(); // Calls Circle's draw()

    delete shape;

    return 0;

}
```

Polymorphism Summary

Type	Technique	Example
Compile-time Polymorphism	Function Overloading	`void show(int); void show(double);`
	Operator Overloading	`Complex operator+(Complex);`
	Templates	`template <typename T> T add(T, T);`
Run-time Polymorphism	Virtual Functions	`virtual void show();`
	Pure Virtual Functions	`virtual void draw() = 0;`

How to Verify Function Overriding in C++?

In C++, **function overriding** occurs when a **derived class** provides a new implementation for a **virtual function** from the **base class**.

Verification Methods:

1. **Using the `override` Specifier (Best Practice)**
2. **Using the `virtual` Keyword**
3. **Checking the VTable Mechanism (Advanced Debugging)**

1. Using the `override` Specifier (Recommended)

The `override` keyword helps verify that a function **actually overrides** a base class function. If there's a mismatch (wrong signature), the **compiler gives an error**.

```
#include <iostream>

class Base {

public:

   virtual void show() { // Virtual function in base class

      std::cout << "Base class function\n";

   }

};

class Derived : public Base {

public:

   void show() override { // Correctly overrides base class function

      std::cout << "Derived class function\n";

   }

};

int main() {

   Base* bptr = new Derived();
```

```
    bptr->show(); // Calls Derived's show() (overridden method)

    delete bptr;

    return 0;

}
```

If there's a mismatch (e.g., different return type, parameters), the compiler throws an error.
Ensures correctness at compile time!

Example of an error :

```
class Derived : public Base {

public:

    void show(int x) override { // ❌ Error: No matching base function!

        std::cout << "Wrong override\n";

    }

};
```

Compiler Error: "show(int)" does not override a function in "Base"
✔ **Fix:** Ensure the signature matches **exactly**.

2. Using the virtual Keyword

Although the `virtual` keyword enables overriding, it **does NOT verify correctness**. If there's a mismatch, **the function will not override** (but no error is shown).

```
#include <iostream>

class Base {

public:

    virtual void show() { // Virtual function in base class

        std::cout << "Base class function\n";

    }

};

class Derived : public Base {
```

```
public:
    void show(int x) { // ❌ Not overriding (no error, but no polymorphism)
        std::cout << "Wrong function, not overriding\n";
    }
};
int main() {
    Base* bptr = new Derived();
    bptr->show(); // Calls Base's show() (not overridden!)
    delete bptr;
    return 0;
}
```

output :Base class function

Problem:

- Since `show(int x)` has a different signature, it **does not override**.
- The `Base` class function **remains active**, leading to **unexpected behavior**.

✅ **Solution:** Use `override` to catch mistakes at compile time.

Is the override Keyword Required in C++?

🔹 **No, the `override` keyword is not required in C++.**
🔹 However, **using `override` is strongly recommended** because it helps catch **mistakes at compile time**.

Without `override` (Potential Mistakes)

If you don't use `override`, the compiler **won't warn you** if there is a mismatch in function signature, leading to **function hiding instead of overriding**.

```
#include <iostream>
class Base {
public:
    virtual void show() { // Virtual function
        std::cout << "Base class function\n";
```

```
    }
};
class Derived : public Base {
public:
    void show(int x) { // ❌ Mistake: Different signature (not overriding)
        std::cout << "Derived class function\n";
    }
};
int main() {
    Base* bptr = new Derived();
    bptr->show(); // Calls Base's function instead of Derived's!
    delete bptr;
    return 0;
}
```

Problem: show(int x) does not override show(), so the base class function is called instead.

With override (Safer)

Using `override`, the compiler will **throw an error** if the function signature does not match.

```
#include <iostream>
class Base {
public:
    virtual void show() { // Virtual function
        std::cout << "Base class function\n";
    }
};
class Derived : public Base {
```

```
public:
    void show(int x) override { // ❌ Compiler Error: No matching base function
        std::cout << "Derived class function\n";
    }
};
int main() {
    Base* bptr = new Derived();
    bptr->show();
    delete bptr;
    return 0;
}
```

error: 'void Derived::show(int)' marked 'override', but does not override any member functions

Key Benefits of Using override

Feature	Without `override`	With `override`
Compiler catches mistakes	❌ No	✅ Yes
Accidental function hiding	✅ Possible	❌ Prevented
Code clarity	❌ Less clear	✅ Explicitly indicates overriding
Required by C++ standard	❌ No	❌ No (but recommended)

✓ override **is not required, but highly recommended.**
✓ Helps **prevent bugs** by catching **incorrect function signatures** at compile time.
✓ Makes the code **easier to read and maintain**.

Would you like to see more **examples of multiple inheritance and** `override` **behavior**?

Copy Constructor, Friend Function, Shallow Copy & Deep Copy in C++

These concepts are crucial in **memory management** and **object copying** in C++.

Copy Constructor

A **copy constructor** is a special constructor that creates a **new object** as a **copy of an existing object**.

Syntax: ClassName(const ClassName& obj);

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

class CopyExample {
public:
   int x;
   // Constructor
   CopyExample(int val) {
      x = val;
   }
   // Copy Constructor
   CopyExample(const CopyExample &obj) {
      x = obj.x;
   }
   void show() {
      cout << "Value: " << x << endl;
   }
};
int main() {
   CopyExample obj1(10);  // Calls normal constructor
   CopyExample obj2 = obj1; // Calls copy constructor
   obj1.show();
   obj2.show();
   return 0;
}
```

Friend Function

A **friend function** is a function that can **access private and protected** members of a class. It is **NOT a member function** but is declared **inside the class using** `friend` **keyword**.

```
#include <iostream>

using namespace std;

class FriendExample {

private:

   int secret;

public:

   FriendExample(int val) : secret(val) {}



   // Friend function declaration

   friend void displaySecret(FriendExample obj);

};

// Friend function definition

void displaySecret(FriendExample obj) {

   cout << "Secret value: " << obj.secret << endl;

}

int main() {

   FriendExample obj(42);

   displaySecret(obj); // Friend function can access private data

   return 0;

}
```

Friend functions allow access to private members without breaking encapsulation.

Shallow Copy vs Deep Copy

When copying objects in C++, **shallow copy** and **deep copy** determine how data is duplicated.

When copying objects in C++, **shallow copy** and **deep copy** determine how data is duplicated.

Feature	Shallow Copy	Deep Copy
Definition	Copies **only references** (pointers)	Copies **actual data**
Memory Usage	Less (shares memory)	More (allocates new memory)
Risk	Multiple objects **point to same memory** (dangerous!)	No shared memory (safe)
Use Case	When object sharing is needed	When unique object copies are needed

3.1 Shallow Copy (Default Copy Constructor)

A **shallow copy** copies object **memory addresses**, causing multiple objects to **share the same memory**.

```
#include <iostream>

using namespace std;

class ShallowCopy {

public:

   int *ptr;

   ShallowCopy(int val) {

      ptr = new int(val);

   }

   // Default copy constructor (shallow copy)
```

```
    void show() {
        cout << "Value: " << *ptr << endl;
    }
};
int main() {
    ShallowCopy obj1(10);
    ShallowCopy obj2 = obj1; // Shallow copy (default constructor)
    *obj2.ptr = 20; // Modifying obj2 affects obj1 too!
    obj1.show();
    obj2.show();
    return 0;
}
```

Problem:

- Both objects share **same memory**.
- Changing `obj2.ptr` **affects** `obj1` **too!**
- This leads to **dangling pointers and crashes** if memory is freed.

3.2 Deep Copy (Custom Copy Constructor)

A **deep copy** creates a **new memory allocation** for the copied object.

```
#include <iostream>
using namespace std;
class DeepCopy {
public:
    int *ptr;
    DeepCopy(int val) {
        ptr = new int(val);
    }
    // Custom deep copy constructor
```

```
    DeepCopy(const DeepCopy &obj) {
        ptr = new int(*obj.ptr); // Allocate new memory
    }
    void show() {
        cout << "Value: " << *ptr << endl;
    }
    ~DeepCopy() {
        delete ptr; // Free allocated memory
    }
};
int main() {
    DeepCopy obj1(10);
    DeepCopy obj2 = obj1; // Deep copy
    *obj2.ptr = 20; // Modifying obj2 does NOT affect obj1
    obj1.show();
    obj2.show();
    return 0;
}
```

Summary Table

Concept	Explanation	Example
Copy Constructor	Creates a new object as a copy of an existing object	`ClassName(const ClassName& obj);`
Friend Function	A function that can access private members of a class	`friend void display(FriendExample obj);`
Shallow Copy	Copies only memory addresses (dangerous!)	`ClassName obj2 = obj1;` (default copy)
Deep Copy	Allocates **new memory** for copied objects	`ptr = new int(*obj.ptr);`

Would you like a **real-world example of deep copy (e.g., dynamic arrays, file handling)?**

Friend Class in C++

A **friend class** in C++ allows one class to **access private and protected members** of another class.

- **Friend class breaks encapsulation but is useful when two classes need close cooperation.**

```
class B; // Forward declaration

class A {

private:

   int secret;

public:

   A() : secret(42) {}

   // Declaring class B as a friend

   friend class B;

};

class B {

public:

   void showSecret(A obj) {

      cout << "Secret: " << obj.secret << endl; // Accessing private member of A

   }

};

int main() {

   A a;

   B b;

   b.showSecret(a); // Output: Secret: 42

   return 0;

}
```

output : Secret :42

Key Characteristics of Friend Class

Feature	Explanation
Access Private Members	A friend class can access private/protected members of another class.
Declared Inside the Class	`friend class ClassName;` is written inside the class whose members need to be accessed.
Breaks Encapsulation	Use carefully, as it allows direct access to private data.

Example: Friend Class in Real-Life Scenario

🛒 E-Commerce Example: `Customer` and `Billing` Classes

```
#include <iostream>

using namespace std;


class Customer {

private:

   string name;

   double balance;


public:

   Customer(string n, double b) : name(n), balance(b) {}


   // Declaring Billing as a friend class

   friend class Billing;

};


class Billing {

public:

   void generateBill(Customer c) {
```

```
        cout << "Customer Name: " << c.name << endl; // Accessing private member
        cout << "Total Amount Due: $" << c.balance << endl;
    }
};

int main() {
    Customer c1("John Doe", 150.75);
    Billing bill;
    bill.generateBill(c1);
    return 0;
}
```

Output:

Customer Name: John Doe Total Amount Due: $150.75

Billing class can access private customer data to generate a bill.

Friend Class vs Friend Function

Feature	Friend Class	Friend Function
Access Level	Can access **all private members**	Can access **only specific members**
Scope	Grants access to **all functions** of a friend class	Grants access to **one function**
Use Case	When multiple methods need access	When only one function needs access

Smart Pointers in C++ (std::unique_ptr, std::shared_ptr, std::weak_ptr)

Smart pointers are part of the C++ Standard Library (`<memory>`). They help manage memory automatically by handling object lifetimes, preventing memory leaks and dangling pointers.

1. std::unique_ptr (Exclusive Ownership)

- Ensures **only one** owner for the allocated object.

- Automatically deletes the object when it goes out of scope.
- Cannot be copied, but can be moved.

```
#include <iostream>

#include <memory>


class MyClass {

public:

   MyClass() { std::cout << "Object Created\n"; }

   ~MyClass() { std::cout << "Object Destroyed\n"; }

   void show() { std::cout << "Hello from MyClass\n"; }

};

int main() {

   std::unique_ptr<MyClass> ptr1 = std::make_unique<MyClass>();

   ptr1->show();

   // std::unique_ptr<MyClass> ptr2 = ptr1; // ❌ ERROR: Cannot copy unique_ptr

   std::unique_ptr<MyClass> ptr2 = std::move(ptr1); // ☑ Transfer ownership

   if (!ptr1) {

      std::cout << "ptr1 is now nullptr\n";

   }

   return 0;

}
```

Use case: When you want exclusive ownership of a resource, such as file handles, sockets, or database connections.

2. std::shared_ptr (Reference Counting)

- Allows **multiple** smart pointers to share ownership of the same object.
- Deletes the object **only when the last** `shared_ptr` **goes out of scope**.

```
#include <iostream>

#include <memory>

class MyClass {

public:

   MyClass() { std::cout << "Object Created\n"; }

   ~MyClass() { std::cout << "Object Destroyed\n"; }

};

int main() {

   std::shared_ptr<MyClass> ptr1 = std::make_shared<MyClass>();

   std::shared_ptr<MyClass> ptr2 = ptr1; // Both ptr1 and ptr2 share ownership

   std::cout << "Reference count: " << ptr1.use_count() << "\n";

   ptr1.reset(); // Releases ptr1, but ptr2 still owns the object

   std::cout << "Reference count after ptr1.reset(): " << ptr2.use_count() << "\n";

   return 0;

}
```

Use case: When multiple parts of the program need to share ownership, such as in caching mechanisms or graph/tree structures.

3. std::weak_ptr (Non-Ownership Reference)

- Works with `shared_ptr` but does **not** increase the reference count.
- Used to avoid **circular references** in graphs or observer patterns.
- Needs to be **converted to** `shared_ptr` before accessing the object.

```
#include <iostream>

#include <memory>

class MyClass {

public:

   MyClass() { std::cout << "Object Created\n"; }

   ~MyClass() { std::cout << "Object Destroyed\n"; }
```

```
};

int main() {

   std::shared_ptr<MyClass> sharedPtr = std::make_shared<MyClass>();

   std::weak_ptr<MyClass> weakPtr = sharedPtr; // Does not increase reference count

   if (auto sp = weakPtr.lock()) { // Convert weak_ptr to shared_ptr safely

      std::cout << "Object is still alive\n";

   } else {

      std::cout << "Object has been deleted\n";

   }

   sharedPtr.reset(); // Deletes object

   if (auto sp = weakPtr.lock()) {

      std::cout << "Object is still alive\n";

   } else {

      std::cout << "Object has been deleted\n";

   }

   return 0;

}
```

Use case: When you need a weak reference to an object without affecting its lifetime (e.g., observer pattern, avoiding memory leaks in cyclic references).

4. Circular Reference Problem with shared_ptr (Solved by weak_ptr)

A **circular reference** occurs when two objects reference each other using `shared_ptr`, preventing proper memory deallocation.

Problem Example:

```
#include <iostream>
```

```
#include <memory>
class B; // Forward declaration
class A {
public:
    std::shared_ptr<B> b_ptr;
    ~A() { std::cout << "A Destroyed\n"; }
};
class B {
public:
    std::shared_ptr<A> a_ptr;
    ~B() { std::cout << "B Destroyed\n"; }
};
int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();
    a->b_ptr = b;
    b->a_ptr = a; // Circular reference
    return 0; // Memory leak! (Objects never get deleted)
}
```

Solution using weak_ptr:

```
#include <iostream>
#include <memory>

class B; // Forward declaration

class A {
```

```
public:
    std::weak_ptr<B> b_ptr; // Use weak_ptr to break cycle
    ~A() { std::cout << "A Destroyed\n"; }
};

class B {
public:
    std::weak_ptr<A> a_ptr; // Use weak_ptr to break cycle
    ~B() { std::cout << "B Destroyed\n"; }
};

int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();

    a->b_ptr = b;
    b->a_ptr = a; // No memory leak!

    return 0; // Objects get properly deleted
}
```

Use case: When objects form a cycle, weak_ptr ensures memory is correctly freed.

Comparison of Smart Pointers

Smart Pointer	Ownership	Reference Count	Can Be Null	Thread-Safe
`unique_ptr`	Exclusive	No	Yes	No
`shared_ptr`	Shared	Yes	Yes	Yes

Smart Pointer	Ownership	Reference Count	Can Be Null	Thread-Safe
`weak_ptr`	No ownership	No	Yes	Yes

When to Use Each?

- ✅ `unique_ptr` → When **only one owner** is needed (`std::make_unique<T>()` is preferred).
- ✅ `shared_ptr` → When **multiple owners** share the same object.
- ✅ `weak_ptr` → To **avoid circular references** and allow **temporary access** to a shared resource.

Data Structures and Algorithms (DSA) in C++

Data Structures and Algorithms (DSA) form the backbone of computer science and programming. **Data structures** help in organizing and managing data, while **algorithms** provide efficient ways to process it.

Data Structures in C++

Data structures store and organize data efficiently.

Type	Definition	Example
Array	Fixed-size collection of elements	`int arr[5] = {1, 2, 3, 4, 5};`
Linked List	Elements (nodes) connected via pointers	Singly, Doubly, Circular
Stack	LIFO (Last In, First Out) data structure	Used in function calls, undo/redo
Queue	FIFO (First In, First Out) structure	Used in scheduling, printing tasks
Deque	Double-ended queue	`std::deque<int> d;`
Priority Queue	Special queue with priorities	`std::priority_queue<int> pq;`
Hash Table	Key-value pair structure	`std::unordered_map<int, string> map;`
Graph	Collection of nodes and edges	Used in networks, shortest path
Heap	Binary heap (Min/Max) for priority management	Used in priority queues
Trie	Tree structure for prefix-based searching	Used in autocomplete, dictionaries

2️⃣ Algorithms in C++

Algorithms define a step-by-step procedure to solve a problem.

Algorithms define a step-by-step procedure to solve a problem.

Sorting Algorithms

Algorithm	Best Time Complexity	Worst Time Complexity	Space Complexity
Bubble Sort	O(n)	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	$O(n^2)$	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(1)

Searching Algorithms

Algorithm	Best Time Complexity	Worst Time Complexity
Linear Search	O(1)	O(n)
Binary Search	O(1)	O(log n)

3 Implementation of Data Structures in C++

(a) Stack using STL

```
#include <iostream>

#include <stack>

using namespace std;


int main() {

   stack<int> s;

   s.push(10);

   s.push(20);

   s.push(30);

   cout << "Top element: " << s.top() << endl; // Output: 30

   s.pop();

   cout << "Top after pop: " << s.top() << endl; // Output: 20

   return 0;

}
```

(b) Queue using STL

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);

    cout << "Front: " << q.front() << endl; // Output: 1
    q.pop();
    cout << "Front after pop: " << q.front() << endl; // Output: 2
    return 0;
}
```

(c) Linked List (Singly)

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

class Node {
public:
    int data;
    Node* next;
    Node(int val) {
        data = val;
```

```
        next = nullptr;
    }
};

class LinkedList {
public:
    Node* head;
    LinkedList() { head = nullptr; }

    void insert(int val) {
        Node* newNode = new Node(val);
        newNode->next = head;
        head = newNode;
    }

    void display() {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};

int main() {
```

```
    LinkedList list;
    list.insert(10);
    list.insert(20);
    list.insert(30);
    list.display(); // Output: 30 -> 20 -> 10 -> NULL
    return 0;
}
```

Important Algorithm Implementations

(a) Binary Search (Recursive)

```
#include <iostream>
using namespace std;
int binarySearch(int arr[], int left, int right, int key) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == key) return mid;
        if (arr[mid] > key) return binarySearch(arr, left, mid - 1, key);
        return binarySearch(arr, mid + 1, right, key);
    }
    return -1;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);

    int key = 30;

    int result = binarySearch(arr, 0, n - 1, key);



    if (result != -1)

        cout << "Element found at index " << result << endl;

    else

        cout << "Element not found\n";



    return 0;

}
```

Quick Sort Algorithm in C++ :

Quick Sort is a divide-and-conquer sorting algorithm that works by selecting a pivot element, partitioning the array around the pivot, and then recursively sorting the left and right subarrays.

Time Complexity:

Case	Complexity
Best Case	**O(n log n)**
Average Case	**O(n log n)**
Worst Case	**$O(n^2)$** (when the smallest or largest element is always chosen as the pivot)

Quick Sort Implementation in C++

```
#include <iostream>

using namespace std;

// Function to swap two elements

void swap(int &a, int &b) {

    int temp = a;

    a = b;

    b = temp;
```

```
}
// Partition function to place pivot at correct position
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as pivot
    int i = low - 1; // Index of smaller element
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) { // If current element is smaller than pivot
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]); // Swap pivot with the correct position
    return i + 1; // Return partition index
}
// Recursive Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Find partition index
        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}
// Driver code
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    printArray(arr, n);
    quickSort(arr, 0, n - 1); // Sorting the array
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0; }
```

2 Explanation of Quick Sort

1. **Select a pivot element** (we choose the last element `arr[high]`).
2. **Partitioning**: Rearrange elements so that:
    - Elements **smaller than pivot** are on the left.
    - Elements **greater than pivot** are on the right.
3. **Recursive Calls**:
    - Apply Quick Sort to the **left subarray** (`low` to `pivot - 1`).
    - Apply Quick Sort to the **right subarray** (`pivot + 1` to `high`).

Example Execution

Input:

```
Original array: 10 7 8 9 1 5
```

Sorting Steps:

1. Choose pivot = **5** → Partition: **[1 5]** | **[10 7 8 9]**
2. Sort left part: **[1]**
3. Sort right part:
    - Choose pivot = **9** → Partition: **[7 8 9]** | **[10]**
    - Sort left part: **[7 8]**
    - Choose pivot = **8** → Partition: **[7 8]**
    - Sort right part: **[10]**

Output:

```
Sorted array: 1 5 7 8 9 10
```

🔼 Time Complexity Analysis

Case	Explanation
Best/Average Case: O(n log n)	Recursively divides the array in half (log n levels), each with O(n) operations
Worst Case: $O(n^2)$	When the smallest or largest element is always chosen as pivot (highly unbalanced partitions)

To improve Quick Sort, use:
✅ **Randomized pivot selection**
✅ **Median-of-three pivot selection**

Memory Leaks in Embedded Systems

Introduction

Memory management is a crucial aspect of embedded system development. Unlike general-purpose computers, embedded systems often operate with **limited memory resources**, making efficient memory utilization essential. A **memory leak** occurs when memory is allocated dynamically but not properly released, leading to **gradual memory depletion** and potential system failures over time. In this blog, we'll explore memory leaks in embedded systems, their causes, impact, detection methods, and best practices to prevent them.

What is a Memory Leak in Embedded Systems?

A memory leak in an embedded system occurs when dynamically allocated memory is not properly freed, causing a slow accumulation of used memory that can never be reclaimed. Over time, this reduces the available memory, leading to **performance degradation, crashes, or system instability**. Unlike desktop or server applications, embedded systems often **lack operating system-level memory management tools**, making memory leaks more difficult to diagnose and fix.

Causes of Memory Leaks in Embedded Systems

Memory leaks in embedded systems can arise due to several factors:

1.Failure to Free Allocated Memory

●In languages like **C and C++**, memory allocated using malloc(), calloc(), or new must be explicitly freed using free() or delete. Forgetting to do so results in a memory leak.

2.Memory Fragmentation

●Continuous allocation and deallocation of different-sized memory blocks lead to fragmentation, making it difficult to allocate large contiguous memory blocks, which can mimic a memory leak.

3.Dangling Pointers and Circular References

●If a pointer referencing allocated memory is lost or overwritten before free() is called, the memory remains allocated but inaccessible.

●In **object-oriented designs**, circular references (e.g., two objects referencing each other) can prevent memory from being released.

4.Unreleased Buffers and Resources

●Buffers, file handles, and I/O resources that are not released properly lead to gradual memory exhaustion.

5.Task and Thread Leaks

●In **real-time operating systems (RTOS)**, tasks, semaphores, and message queues that are not properly deleted can accumulate over time, consuming memory.

6.Persistent Global Variables and Static Allocations

●Improper use of global variables and statically allocated structures can lead to excessive memory consumption over time.

Impact of Memory Leaks in Embedded Systems

Memory leaks can have severe consequences, including:

●**Gradual Performance Degradation**: Available memory decreases over time, causing slower execution and delayed responses.

●**Unexpected Crashes and Resets**: As memory becomes exhausted, the system may crash or trigger a **watchdog timer reset**.

●**Increased Power Consumption**: More CPU cycles spent managing fragmented memory result in higher power usage.

●**System Instability in Critical Applications**: Memory leaks in **automotive, medical, or industrial control systems** can lead to catastrophic failures.

Detecting Memory Leaks in Embedded Systems

Detecting memory leaks in embedded systems is challenging due to limited debugging tools, but several methods can help:

1. Static Code Analysis

●Tools like **Cppcheck**, **Lint**, and **Clang Static Analyzer** help detect memory mismanagement issues before execution.

2. Runtime Memory Profiling

●Using **RTOS memory tracking APIs**, developers can monitor heap usage and detect unusual memory growth.

●**Valgrind**, **Electric Fence**, and **AddressSanitizer** are tools that help analyze memory behavior in some embedded environments.

3. Heap Usage Monitoring

●Many RTOS (like FreeRTOS) provide functions such as uxTaskGetStackHighWaterMark() and xPortGetFreeHeapSize() to track memory usage.

4. Leak Detection in Debug Mode

●Custom **debug malloc/free wrappers** can log memory allocations and detect leaks.

5. Hardware Debugging Tools

●**JTAG debuggers, emulators, and logic analyzers** can help in real-time memory monitoring and debugging.

Preventing Memory Leaks in Embedded Systems

Preventing memory leaks requires following best practices for memory management:

1. Minimize Dynamic Memory Allocation

●Prefer **static allocation** over malloc() or new to avoid fragmentation and leaks.

2. Use Memory Pools

●Implement **fixed-size memory pools** instead of relying on the heap for dynamic allocations.

3. Implement Proper Resource Management

●Always **free allocated memory** and release file handles, buffers, and I/O resources after use.

4. Use Smart Pointers (C++)

●Utilize **std::unique_ptr and std::shared_ptr** to automatically manage memory allocation and deallocation.

5. Monitor and Log Memory Usage

●Implement **heap usage logging** to track memory trends and detect leaks before they become critical.

6. Perform Code Reviews and Testing

●Regular **peer code reviews**, stress tests, and **automated static analysis** help detect memory leaks early.

Conclusion

Memory leaks in embedded systems are particularly dangerous due to **limited memory resources and lack of advanced memory management tools**. Identifying and fixing memory leaks requires **careful coding practices, runtime monitoring, and static analysis tools**. By following best practices such as **minimizing dynamic allocations, using memory pools, and monitoring heap usage**, developers can create reliable, efficient, and stable embedded systems.

https://yoginsavani.com/memory-leaks-in-embedded-systems/#google_vignette

```
#include <stdio.h>

#include <stdint.h>

//swap bit 17 with bit 25 in a 64bit integer  ==> 17 =0 25 =1;

 ********** Bit swap in c **********

int main()

{

   uint64_t a =0x700ae, temp1, temp2; // 000{0} 0000 011{1} 0000 0000 1010 1110

   printf("%lx\n", a);

   int swap_bit1, swap_bit2 ;

   swap_bit1 = ( a>>17) & 1;

   swap_bit2 = (a >> 25) & 1;
```

```
    printf("swap_bit1 :%x\n", swap_bit1);
    printf("swap_bit2 :%x\n", swap_bit2);
    if ( swap_bit2 == swap_bit1)
       {
          printf("%lx \n", a);
          return 0 ;
       }
    else
    {
       //a = (a^(1 << 25));   // this two line ok - can below and this line code
       //a = (a^(1 << 17));   // this two line ok
       a ^= (1 << 25) | (1 << 17);  // this is also fine
    }

    printf("%lx \n", a);
    return 0;
}
****************** Bit swap in c++ ******************************
#include <iostream>
#include <bitset>
int main()
{
    uint64_t a = 0x700ae; // 0000 0000 0000 0111 0000 0000 1010 1110
    printf("Before swap: %lx\n", a);
```

```
    int swap_bit1 = (a >> 17) & 1;  // Extract bit 17
    int swap_bit2 = (a >> 25) & 1;  // Extract bit 25
    printf("Bit 17: %x\n", swap_bit1);
    printf("Bit 25: %x\n", swap_bit2);
    // If the bits are different, toggle them
    if (swap_bit1 != swap_bit2)
    {
        a ^= (1ULL << 17);  // Toggle bit 17
        a ^= (1ULL << 25);  // Toggle bit 25
    }
    printf("After swap: %lx\n", a);
    return 0;
}
****************** Bit modification code *******************************
#include <stdio.h>
#include <stdint.h>
// bit modification code
int main()
{
    uint64_t a = 0x70;
    printf("%lx \n", a);
    int kth = 2 ;  // kth location of the bit
    //  a= a | (1 << kth);      // set the kth bit or make a kth bit
    //  a = a & ~(1 << kth);    // reset the kth bit or clearing a kth bit
```

```
    //  a = a ^ (1 << kth);     // Toggle the kth bit  or modifying the bit
    //  a = (a>> kth) & 1 ;     // Find the kth bit
    printf("%lx \n", a);
    return 0;
}
```

**********Hex number swap **********

```
#include <stdio.h>
// Function to swap two nibbles in a 16-bit number
unsigned short swap_nibbles(unsigned short num, int pos1, int pos2) {
    // Extract nibbles (each nibble is 4 bits)
    unsigned short nibble1 = (num >> (pos1 * 4)) & 0xF;  // Extract nibble at pos1
    unsigned short nibble2 = (num >> (pos2 * 4)) & 0xF;  // Extract nibble at pos2
    // Clear the original nibble positions
    num &= ~(0xF << (pos1 * 4));  // Clear nibble1 position
    num &= ~(0xF << (pos2 * 4));  // Clear nibble2 position
    // Swap and set the new values
    num |= (nibble1 << (pos2 * 4));  // Place nibble1 at pos2
    num |= (nibble2 << (pos1 * 4));  // Place nibble2 at pos1
    return num;
}
int main() {
    unsigned short num = 0x9876;
```

```
    // Swap 2nd nibble (8) at position 3 with 4th nibble (6) at position 0
    unsigned short swapped = swap_nibbles(num, 3, 0);
    printf("Original: 0x%X\n", num);
    printf("Swapped:  0x%X\n", swapped);  // Expected output: 0x9678
    return 0;
}
```

**********Decimal number swap **********

```
#include <stdio.h>
// Function to swap two digits in a 4-digit number
int swap_digits(int num, int pos1, int pos2) {
    // Extract digits at given positions
    int digit1 = (num / pos1) % 10;
    int digit2 = (num / pos2) % 10;
    // Clear original digit positions
    num -= digit1 * pos1;  // Remove digit1
    num -= digit2 * pos2;  // Remove digit2
    // Swap and set new values
    num += digit1 * pos2;  // Place digit1 at pos2
    num += digit2 * pos1;  // Place digit2 at pos1
    return num;
}
int main() {
    int num = 9876;
```

```
    // Swap 100's place (8, at position 100) with 1's place (6, at position 1)
    int swapped = swap_digits(num, 100, 1);
    printf("Original: %d\n", num);
    printf("Swapped: %d\n", swapped);  // Expected output: 9678
    return 0;
}
```

dead lock , mutex, semaphores, spinlock

- ✅ **View processes:** `ps`, `top`, `htop`
- ✅ **Manage processes:** `kill`, `nice`, `renice`
- ✅ **Control foreground/background:** `fg`, `bg`, `jobs`
- ✅ **Create processes:** `fork()`, `exec()`
- ✅ **Handle orphans/zombies:** `init`, `kill`