Object-Oriented Programming Interview Preparation Guide

Core OOP Concepts

- **Class**: Blueprint for creating objects; defines attributes and behaviors
- **Object**: Instance of a class; has state and behavior
- **Data Members**: Variables that belong to a class/object
- **Member Functions**: Functions that belong to a class/object
- **Constructor**: Special method called during object creation; initializes object state
- **Destructor**: Special method called when object is destroyed; releases resources (crucial in C++)

Four Pillars of OOP

1. Encapsulation

- **Definition**: Bundling data and methods that operate on that data within a single unit
- **Implementation**: Private data members with public getter/setter methods
- **Benefits**: Information hiding, control over data access, reduces system complexity
- **Example**: `class BankAccount { private: double balance; public: void deposit(double amount) { /* checks and updates */ } }`

2. Inheritance

- **Definition**: Mechanism where a new class derives properties/methods from existing class
- **Base/Parent Class**: Class being inherited from
- **Derived/Child Class**: Class that inherits
- **Types**:
  - **Single**: One child, one parent
  - **Multiple**: One child, multiple parents (supported in C++, unlike Java)
  - **Multilevel**: Child becomes parent for another class
  - **Hierarchical**: One parent, multiple children
- **Benefits**: Code reuse, establishes "is-a" relationships
- **Example**: `class SavingsAccount : public BankAccount { /* additional features */ }`

3. Polymorphism

- **Definition**: Ability of objects to take different forms depending on context
- **Types**:
  - **Compile-time/Static**: Function overloading, operator overloading

- **Runtime/Dynamic**: Function overriding using virtual functions

- **Benefits**: Flexibility, extensibility, simplifies code
- **Example**: `void draw(Circle c)` and `void draw(Square s)` (overloading) or `shape->draw()` works differently based on actual object type (overriding with virtual functions)

4. Abstraction

- **Definition**: Hiding implementation details, showing only functionality
- **Implementation**: Abstract classes (containing pure virtual functions)
- **Benefits**: Simplifies complex systems, focuses on what not how
- **Example**: `class Shape { public: virtual void draw() = 0; }` - defines action without implementation

Abstract Class vs Interface

- **Abstract Class**:
  - Has at least one pure virtual function
  - Can have concrete and pure virtual functions
  - Can have data members and constructors
  - Cannot be instantiated
  - Supports multiple inheritance in C++
  - Use when "is-a" relationship
- **Interface**:
  - Not a native C++ construct (unlike Java)
  - Implemented as abstract class with only pure virtual functions
  - No data members (except static const)
  - No method implementations
  - Common convention: `class IDrawable { public: virtual void draw() = 0; virtual ~IDrawable() {} };`

Function Overloading vs Overriding

- **Overloading**:
  - Same function name, different parameters
  - Happens in same class
  - Compile-time polymorphism
  - Example: `add(int, int)` and `add(float, float)`

- **Overriding**:
  - Same function signature in base and derived class
  - Runtime polymorphism (requires virtual in C++)
  - Requires inheritance
  - Example: `Animal::makeSound()` overridden in `Dog::makeSound()`
  - Use `override` keyword (C++11) to ensure proper overriding

Association, Aggregation, and Composition

- **Association**: Relationship between two classes (uses-a)
  - Example: Teacher teaches Student
- **Aggregation**: Weak "has-a" relationship (part can exist independently)
  - Example: `Department` has pointers to `Professor` objects
- **Composition**: Strong "has-a" relationship (part cannot exist independently)
  - Example: `House` contains `Room` objects directly, not as pointers

Static vs Non-static

- **Static**:
  - Belongs to class, not objects
  - Shared among all instances
  - Accessed using class name
  - In C++, static data members need separate definition outside class
  - Example: `Math::PI`, utility methods
- **Non-static**:
  - Belongs to object instances
  - Each object has its own copy
  - Access requires object instance
  - Example: object state variables

Access Specifiers

- **Public**: Accessible from anywhere
- **Protected**: Accessible within class, derived classes, and friend functions
- **Private**: Accessible only within class and friend functions

SOLID Principles

- **S - Single Responsibility**: A class should have only one reason to change
- **O - Open/Closed**: Open for extension, closed for modification
- **L - Liskov Substitution**: Subtypes must be substitutable for their base types
- **I - Interface Segregation**: Many client-specific interfaces better than one general-purpose interface
- **D - Dependency Inversion**: Depend on abstractions, not concretions

Design Patterns

Creational Patterns

- **Singleton**: Ensures class has only one instance
- **Factory Method**: Creates objects without specifying exact class
- **Abstract Factory**: Creates families of related objects
- **Builder**: Constructs complex objects step by step
- **Prototype**: Creates new objects by copying existing ones

Structural Patterns

- **Adapter**: Allows incompatible interfaces to work together
- **Decorator**: Adds responsibilities to objects dynamically
- **Facade**: Provides simplified interface to complex subsystem
- **Composite**: Treats group of objects as single object
- **Proxy**: Object representing another object

Behavioral Patterns

- **Observer**: One-to-many dependency between objects
- **Strategy**: Defines family of algorithms, makes them interchangeable
- **Command**: Encapsulates request as an object
- **Iterator**: Accesses elements sequentially without exposing structure
- **State**: Object behavior changes based on its state

Exception Handling

- **C++ Exception Handling**: try, catch, throw
- **Standard Exceptions**: std::exception as base class
- **Custom Exceptions**: Deriving from std::exception
- **RAII Pattern**: Resource Acquisition Is Initialization for safe resource management
- **noexcept**: Specifies a function won't throw exceptions (C++11)

C++ Specific Features

- **Operator Overloading**: Customizing operator behavior
- **Multiple Inheritance**: Can inherit from multiple base classes
- **Friend Functions/Classes**: Non-member functions/classes accessing private members
- **Virtual Destructors**: Essential for proper cleanup in polymorphic hierarchies
- **Memory Management**: new/delete, smart pointers (unique_ptr, shared_ptr)

Common Interview Questions

Basic OOP

1. **Explain OOP with real-world example?** Car class with attributes (color, model) and methods (accelerate, brake).
2. **What is virtual function with example?** Base class function that can be overridden; e.g., `virtual void speak()` in Animal, overridden in Dog.
3. **Why do we need virtual destructors?** To ensure proper destruction of derived objects through base class pointers.

Inheritance and Polymorphism

1. **When to use abstract class vs interface in C++?** Abstract class: partial implementation, "is-a". Interface: pure protocol, "can-do" (still implemented as abstract class in C++).
2. **Runtime vs compile-time polymorphism?** Runtime: virtual functions; Compile-time: function overloading, templates.
3. **How does C++ handle the diamond problem?** Using virtual inheritance: `class D: public virtual B, public virtual C`.

SOLID and Design

1. **Explain dependency injection in C++?** Providing object dependencies externally rather than creating internally.
2. **How to implement Singleton pattern in C++?** Private constructor, deleted copy/move operations, static method returning static instance.
3. **Real-world example of Strategy pattern?** Different compression algorithms implemented with common interface.

Advanced Concepts

1. **What is RAII in C++?** Resource Acquisition Is Initialization; ensures resources are properly released.
2. **Difference between composition and inheritance?** Composition: "has-a" (more flexible); Inheritance: "is-a" (tighter coupling).

3. **What is object slicing?** Losing derived class attributes when assigning to base class object (not just a pointer or reference).

C++ Specific OOP Features

- **const correctness**: Using const for immutable objects and methods
- **References vs Pointers**: References cannot be null, pointers can
- **Copy/Move Semantics**: Deep copying and efficient resource transfer
- **Rule of Three/Five/Zero**: Guidelines for resource management
- **Templates**: Generic programming construct for type-independent code

Common OOP Pitfalls

- **Deep inheritance hierarchies**: Difficult to understand and maintain
- **God objects**: Classes that know/do too much
- **Tight coupling**: Excessive dependencies between classes
- **Premature abstraction**: Creating hierarchies before understanding domain
- **Memory leaks**: Not properly managing dynamic memory (crucial in C++)
- **Missing virtual destructors**: Leading to resource leaks
- **Slicing**: Object slicing in C++ when using values instead of pointers/references