

OBJECT ORIENTED PROGRAMMING (C++)

Chapter 4: Inheritance

Exam-Oriented Questions and Answers (Short, Broad & Coding)

Section A: Short Answer Type Questions

The following questions are typically asked for 2 to 3 marks. Answers are kept precise and to the point, suitable for quick revision before exams.

Q1. Define inheritance. [2 Marks]

Ans. Inheritance is an OOP mechanism in which one class (derived class) acquires the properties and behaviours (data members and member functions) of another already existing class (base class), enabling code reusability and hierarchical classification.

Q2. What is a base class and a derived class? [2 Marks]

Ans. A base class (or super class) is the existing class whose members are inherited. A derived class (or sub class) is the new class that inherits members from the base class and may add its own additional members.

Q3. Why is the protected access specifier needed when private and public already exist? [2 Marks]

Ans. private members are inaccessible to derived classes, and public members are exposed to the whole program. protected fills the gap - it keeps members hidden from the outside world but still lets derived classes access and modify them directly, which is essential for extending base class functionality.

Q4. What is multilevel inheritance? Give an example. [2 Marks]

Ans. Multilevel inheritance is when a class is derived from another class which is itself derived from a base class, forming a chain. Example: class A (base) -> class B : public A -> class C : public B, so C inherits from B and indirectly from A.

Q5. Differentiate between multiple inheritance and hierarchical inheritance in one line each. [2 Marks]

Ans. Multiple inheritance: one derived class inherits from two or more base classes. Hierarchical inheritance: two or more derived classes inherit from a single common base class.

Q6. What causes ambiguity in multiple inheritance? [2 Marks]

Ans. Ambiguity occurs when a derived class inherits from two or more base classes that contain a member (function or data) with the same name, so the compiler cannot decide which base class's member the derived class object refers to.

Q7. How is ambiguity in multiple inheritance resolved? [2 Marks]

Ans. It is resolved by using the scope resolution operator to explicitly specify the base class (e.g., `obj.BaseName::member()`), or by redefining the member in the derived class itself so that the derived version takes precedence.

Q8. What is a virtual base class? [2 Marks]

Ans. A virtual base class is a base class declared with the `virtual` keyword in an intermediate class, so that when it is inherited through multiple paths in hybrid inheritance, only a single shared copy of it is included in the final derived class, avoiding duplicate-copy ambiguity (the diamond problem).

Q9. What is a pure virtual function? [2 Marks]

Ans. A pure virtual function is a virtual function declared in a base class with no body, written as `virtual returnType functionName() = 0`; It forces derived classes to provide their own implementation and makes the base class abstract.

Q10. Why can an abstract class not be instantiated? [2 Marks]

Ans. An abstract class contains at least one pure virtual function with no implementation, so the compiler cannot create a complete, callable object of it. It can only be used as a base class, and pointers/references of its type can point to objects of concrete derived classes.

Q11. In what order are base and derived class constructors called when a derived object is created? [2 Marks]

Ans. The base class constructor is always called first, followed by the derived class constructor, ensuring the inherited part of the object is initialized before the derived-specific part.

Q12. In what order are base and derived class destructors called? [2 Marks]

Ans. In exactly the reverse order of construction: the derived class destructor executes first, followed by the base class destructor.

Q13. What is containership? [2 Marks]

Ans. Containership (or composition) is a relationship where an object of one class is declared as a data member inside another class, so the outer class 'contains' the inner class's object as one of its parts.

Q14. Differentiate between IS-A and HAS-A relationships in one line each. [2 Marks]

Ans. IS-A relationship is modelled using inheritance and means the derived class is a specialized type of the base class (e.g., Dog IS-A Animal). HAS-A relationship is modelled using containership and means a class owns/contains an object of another class as a part (e.g., Car HAS-A Engine).

Q15. What is a namespace and why is it used? [2 Marks]

Ans. A namespace is a declarative region that groups identifiers (classes, functions, variables) under a distinct name to avoid naming conflicts when large programs combine code from multiple sources; members are accessed using the scope resolution operator, e.g., `NamespaceName::member`.

Q16. What is a friend function? [2 Marks]

Ans. A friend function is a non-member function that is granted special permission, using the `friend` keyword inside the class, to access the private and protected members of that class.

Q17. What is a friend class? [2 Marks]

Ans. A friend class is a class that is granted access to the private and protected members of another class by declaring it as friend class ClassName; inside that class; every member function of the friend class then gets this access.

Q18. State one advantage and one disadvantage of using friend functions. [2 Marks]

Ans. Advantage: a friend function allows controlled, selective access to a class's private data without making it fully public (useful, for example, when overloading operators like <<). Disadvantage: it weakens encapsulation and increases coupling, since an outside function now depends directly on the class's internal representation.

Section B: Broad / Long Answer Type Questions

The following questions are typically asked for 5 to 10 marks and expect explanation with diagrams/examples where relevant.

Q1. What is inheritance? Explain its types with suitable diagrams and C++ syntax. [8 Marks]

Ans. Inheritance is the OOP mechanism by which a derived class acquires the properties and behaviours of a base class, enabling code reuse and hierarchical design. C++ supports the following five types:

- Single Inheritance - one derived class inherits from exactly one base class. Syntax: `class Derived : public Base { };`
- Multilevel Inheritance - a chain of derivation where a class derived from a base class itself acts as a base class for another class (`A -> B -> C`).
- Multiple Inheritance - one derived class inherits from two or more base classes simultaneously: `class C : public A, public B { };`
- Hierarchical Inheritance - multiple derived classes inherit independently from one common base class.
- Hybrid Inheritance - a combination of two or more of the above types in the same program, often leading to the diamond problem, which is resolved using virtual base classes.

Each type serves different design needs - single and multilevel model simple 'is-a' chains, hierarchical models a family of related sub-types sharing common behaviour, and multiple/hybrid combine features from unrelated or overlapping class hierarchies.

Q2. What is the diamond problem in multiple inheritance? Explain with a program how virtual base classes solve it. [8 Marks]

Ans. When two classes B and C are both derived from a common base class A, and a fourth class D inherits from both B and C, D ends up containing two separate copies of A - one via B and one via C. Any attempt to access a member of A through D becomes ambiguous, since the compiler cannot tell which copy is meant. This duplication, drawn as a diamond shape (A at top, B and C in the middle, D at the bottom), is called the diamond problem.

```
class A { public: int x; };
class B : virtual public A { };
class C : virtual public A { };
class D : public B, public C { };

int main() {
    D obj;
    obj.x = 5;    // OK - only one copy of A exists, no ambiguity
}
```

By declaring A as a virtual base class in both B and C, C++ guarantees that only one shared sub-object of A is created inside D, regardless of how many inheritance paths lead to it. This removes the duplicate copies and, with them, the ambiguity.

Q3. Explain abstract classes and pure virtual functions with an example program. [8 Marks]

Ans. An abstract class is a class that cannot be instantiated directly and exists only to be used as a base class. It becomes abstract the moment it contains one or more pure virtual functions - virtual functions declared with '= 0' and no body in the base class. Derived classes must override every pure virtual function to become concrete (instantiable); otherwise they remain abstract too.

```
class Shape          // abstract class
{
public:
    virtual double area() = 0;    // pure virtual function
};

class Rectangle : public Shape
{
    double l, b;
public:
    Rectangle(double l1, double b1) : l(l1), b(b1) {}
    double area() override { return l * b; }
};

int main() {
    // Shape s;          // ERROR: cannot instantiate abstract class
    Shape* s = new Rectangle(4, 5);
    cout << s->area(); // 20 - resolved via runtime polymorphism
}
```

Abstract classes are widely used to define a common interface (such as area() here) that every derived class must implement in its own way, while still allowing the program to work with base-class pointers/references for runtime polymorphism.

Q4. Explain the order of execution of constructors and destructors in inheritance with a program. [6 Marks]

Ans. When a derived class object is created, the base class constructor always runs first, followed by the derived class constructor, since the derived part of the object logically depends on the base part being ready first. Destruction happens in the exact reverse order - the derived class destructor runs first, then the base class destructor.

```
class Base
{
public:
    Base() { cout << "Base constructor" << endl; }
    ~Base() { cout << "Base destructor" << endl; }
};

class Derived : public Base
{
public:
    Derived() { cout << "Derived constructor" << endl; }
    ~Derived() { cout << "Derived destructor" << endl; }
};

int main() {
    Derived obj;
}

/* Output:
Base constructor
Derived constructor
Derived destructor
Base destructor
*/
```

```
Derived destructor  
Base destructor  
*/
```

Q5. Differentiate between IS-A and HAS-A relationships with examples and code. [6 Marks]

Basis	IS-A (Inheritance)	HAS-A (Containership)
Meaning	Derived class is a specialised kind of base class	Class owns/contains an object of another class
C++ mechanism	class Derived : public Base { };	Embedding an object as a data member
Example	Dog IS-A Animal -> class Dog : public Animal	Car HAS-A Engine -> class Car { Engine e; };
Access style	Inherited public/protected members used directly	Member object's functions accessed via dot operator

In short, inheritance models a 'type-of' relationship suited to extending and specialising behaviour, while containership models a 'part-of' or 'has' relationship suited to building complex objects out of simpler, independent ones.

Q6. What are friend functions and friend classes? Discuss their advantages and disadvantages. [8 Marks]

Ans. A friend function is a non-member function granted access to a class's private and protected members by declaring it with the friend keyword inside the class. A friend class is an entire class granted the same access, so all of its member functions can read a private members of the class that befriended it.

```
class Box  
{  
    int length;  
public:  
    Box(int l) : length(l) {}  
    friend void showLength(Box b);    // friend function declaration  
};  
  
void showLength(Box b) { cout << b.length; }    // defined outside, no Box::
```

Advantages: friends allow controlled, selective sharing of private data (useful for operator overloading and functions that must operate on two different classes at once) without fully exposing the data as public. Disadvantages: friends break strict encapsulation, increase coupling between classes (a change inside the class can force changes in every friend), and friendship is neither inherited, symmetric, nor transitive, which limits its flexibility and is generally considered an exception to good OOP design rather than the norm.

Q7. What is a nested class and how is containership different from it? [6 Marks]

Ans. A nested class is a class declared inside another (enclosing) class - it is the class definition itself that lives inside another class, and it is typically used when the inner class exists purely to support the outer class. Containership, on the other hand, is not about where a class is declared; it is about an object (instance) of one already-existing class being embedded as a data member of another class. In short: nesting is about scope of declaration, containership is about composition of objects.

```
class Outer
{
public:
    class Inner { public: void show() { cout << "Inner"; } };    // nested class
};

class Engine { public: int hp; };
class Car { Engine e; };    // containership - Engine object embedded in Car
```

Section C: Coding / Programming Type Questions

The following are typical program-writing questions from this chapter, each with a complete, compilable C++ solution and a short explanation.

Q1. Write a C++ program to demonstrate single inheritance, where a class Student inherits from a class Person. [6 Marks]

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

class Person
{
protected:
    string name;
public:
    void setName(string n) { name = n; }
};

class Student : public Person
{
    int roll;
public:
    void setRoll(int r) { roll = r; }
    void display()
    {
        cout << "Name: " << name << ", Roll: " << roll << endl;
    }
};

int main()
{
    Student s;
    s.setName("Rahul");
    s.setRoll(101);
    s.display();
    return 0;
}
```

Ans. Here, Student inherits publicly from Person, so it directly reuses the protected member 'name' and the setName() function without redefining them, while adding its own member 'roll'. This demonstrates code reuse through single inheritance.

Q2. Write a program to demonstrate multiple inheritance and resolve the resulting ambiguity using the scope resolution operator. [8 Marks]

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

class Teacher
{
public:
    void department() { cout << "Teaches Computer Science" << endl; }
};

class Coach
```

```

{
public:
    void department() { cout << "Coaches Basketball team" << endl; }
};

class TeachingAssistant : public Teacher, public Coach
{
};

int main()
{
    TeachingAssistant ta;
    // ta.department();           // ERROR: ambiguous call
    ta.Teacher::department();     // resolved using scope resolution
    ta.Coach::department();
    return 0;
}

```

Ans. Both Teacher and Coach define department(), so a plain ta.department() call is ambiguous. Prefixing the call with the base class name and the scope resolution operator (Teacher:: or Coach::) tells the compiler exactly which version to invoke.

Q3. Write a program demonstrating the diamond problem and its solution using virtual base classes. [8 Marks]

```

#include <iostream>
using namespace std;

class Person
{
public:
    string name = "Ankita";
};

class Employee : virtual public Person { };
class Volunteer : virtual public Person { };

class CommunityWorker : public Employee, public Volunteer
{
};

int main()
{
    CommunityWorker cw;
    cout << cw.name;    // OK - only one copy of Person exists
    return 0;
}

```

Ans. Without the virtual keyword, CommunityWorker would inherit two separate copies of Person (via Employee and via Volunteer), making cw.name ambiguous. Declaring Person as a virtual base class in both Employee and Volunteer ensures CommunityWorker contains only a single shared copy of Person, so the member access compiles and runs without ambiguity.

Q4. Write a program to demonstrate an abstract class using a pure virtual function for calculating area of different shapes. [8 Marks]

```

#include <iostream>
using namespace std;

```

```

class Shape
{
public:
    virtual double area() = 0;    // pure virtual function -> Shape is abstract
};

class Circle : public Shape
{
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() override { return 3.14159 * radius * radius; }
};

class Square : public Shape
{
    double side;
public:
    Square(double s) : side(s) {}
    double area() override { return side * side; }
};

int main()
{
    Shape* shapes[2];
    shapes[0] = new Circle(5);
    shapes[1] = new Square(4);
    for (int i = 0; i < 2; i++)
        cout << "Area: " << shapes[i]->area() << endl;
    return 0;
}

```

Ans. Shape cannot be instantiated because area() is pure virtual, but a Shape pointer can still point to Circle or Square objects. Calling shapes[i]->area() invokes the correct overridden version for each object at runtime, illustrating both abstract classes and runtime polymorphism.

Q5. Write a program to demonstrate containership (HAS-A relationship) using a Car class that contains an Engine object. [6 Marks]

```

#include <iostream>
using namespace std;

class Engine
{
    int horsepower;
public:
    Engine(int hp) : horsepower(hp)
    {
        cout << "Engine created with " << horsepower << " HP" << endl;
    }
    void showPower() { cout << "Power: " << horsepower << " HP" << endl; }
};

class Car
{
    Engine engine;    // containership - Engine object embedded in Car
    string model;
public:
    Car(string m, int hp) : engine(hp), model(m)
    {
        cout << "Car " << model << " created" << endl;
    }
}

```

```

    void display() { cout << model << " -> "; engine.showPower(); }
};

int main()
{
    Car c("Sedan X", 150);
    c.display();
    return 0;
}

```

Ans. The Engine object is a data member of Car, so 'Car HAS-A Engine'. Notice the constructor initializer list `Car(...) : engine(hp), model(m)` - the embedded Engine object is constructed before the Car constructor's own body runs, which is why 'Engine created...' prints before 'Car Sedan X created'.

Q6. Write a program that uses a friend function to add the private data members of two different classes. [8 Marks]

```

#include <iostream>
using namespace std;

class ClassB;    // forward declaration

class ClassA
{
    int a;
public:
    ClassA(int x) : a(x) {}
    friend int addValues(ClassA, ClassB);    // friend of ClassA
};

class ClassB
{
    int b;
public:
    ClassB(int y) : b(y) {}
    friend int addValues(ClassA, ClassB);    // friend of ClassB
};

int addValues(ClassA objA, ClassB objB)
{
    return objA.a + objB.b;    // accessing private members of both classes
}

int main()
{
    ClassA oa(10);
    ClassB ob(25);
    cout << "Sum = " << addValues(oa, ob);
    return 0;
}

```

Ans. `addValues()` is not a member of either class, yet it is declared as a friend in both `ClassA` and `ClassB`, which lets it directly read the private members 'a' and 'b' of the two unrelated classes and return their sum - a task that would otherwise need public getter functions in both classes.