

# OBJECT ORIENTED PROGRAMMING (C++)

## Chapter 4: Inheritance

*Exam-Oriented Questions and Answers - Set 2 (Short & Broad)*

---

### 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. This set covers additional angles not included in the first short-answer set.

**Q1. How does inheritance support the concept of code reusability? [2 Marks]**

**Ans.** Once common data members and member functions are written in a base class, every derived class automatically gets access to them without rewriting that code. New classes only need to add what is genuinely new or different, so existing, tested code is reused rather than duplicated.

**Q2. What is meant by a 'sub-object' in the context of inheritance? [2 Marks]**

**Ans.** When a derived class object is created, it internally contains a smaller object made up of the inherited base class members. This internally embedded portion is called the base class sub-object, and it is constructed first, before the derived-specific members.

**Q3. Can constructors be inherited by a derived class in C++? [2 Marks]**

**Ans.** No, constructors are not inherited in the ordinary sense - a derived class must define its own constructor if it needs one. However, the base class constructor is still automatically invoked whenever a derived class object is created, either implicitly (default constructor) or explicitly through the initializer list.

**Q4. What access do protected members of a base class get in a derived class under public inheritance? [2 Marks]**

**Ans.** Under public inheritance, protected members of the base class remain protected in the derived class - they stay inaccessible to outside code but continue to be accessible to the derived class's own member functions and to any further classes derived from it.

**Q5. Give one real-world example each of hybrid inheritance. [2 Marks]**

**Ans.** A TeachingAssistant class that inherits from both Teacher and Student (multiple inheritance), where Teacher and Student themselves are both derived from a common Person base class (hierarchical inheritance), is a hybrid inheritance example combining the two forms.

**Q6. State one key difference between a virtual function and a pure virtual function. [2 Marks]**

**Ans.** A virtual function has a body/definition in the base class, which a derived class may optionally override. A pure virtual function has no body in the base class (declared with '= 0'), MUST be overridden by any derived class that wants to be instantiable, and its presence makes the base class abstract.

**Q7. In one line, what is the diamond problem? [2 Marks]**

**Ans.** The diamond problem is the ambiguity that arises when a class inherits from two classes that both derive from the same common base class, resulting in two duplicate copies of that common base inside the final derived class.

**Q8. What role does the scope resolution operator (::) play in resolving inheritance-related ambiguity? [2 Marks]**

**Ans.** It lets the programmer explicitly specify which base class's member is intended (e.g., `obj.BaseName::function()`), removing the compiler's confusion when two or more inherited base classes define a member with the same name.

**Q9. By default, is inheritance from a struct public or private in C++? How does this differ from class? [2 Marks]**

**Ans.** By default, a struct inherits publicly from its base, whereas a class inherits privately from its base if no access specifier is written. This is one of the few practical differences between struct and class in C++ (their members also default to public vs private in the same way).

**Q10. Can a derived class have more than one direct base class? What is this called? [2 Marks]**

**Ans.** Yes, a derived class can directly inherit from two or more base classes at the same time; this is called multiple inheritance, written as `class Derived : public Base1, public Base2 { ;`

**Q11. What is the main structural difference between a nested class and inheritance? [2 Marks]**

**Ans.** A nested class is declared inside another class purely for scoping/organisation and does not automatically give the enclosing class any of its members. Inheritance, by contrast, is a relationship between two separately declared classes where the derived class actively acquires the base class's members.

**Q12. Apart from avoiding name conflicts, what other benefit do namespaces provide? [2 Marks]**

**Ans.** Namespaces also help organise large codebases into logical, named groups (for example, grouping all classes and functions belonging to one library or module), making the code easier to navigate, and allowing the same short identifier names to be reused safely across independent modules.

**Q13. Why is it useful to have a base class pointer point to a derived class object? [2 Marks]**

**Ans.** It allows a single pointer type (and a single array or function parameter of that type) to work with objects of many different derived classes, and, when combined with virtual functions, lets the correct derived class version of a function be called automatically at runtime - the basis of runtime polymorphism.

**Q14. What is protected inheritance, and how does it differ from public and private inheritance? [2 Marks]**

**Ans.** In protected inheritance, both the public and protected members of the base class become protected members of the derived class. This is stricter than public inheritance (which preserves public members as public) but looser than private inheritance (which makes everything private), and it is used when further derived classes still need access but the outside world should not.

**Q15. What happens if a derived class does not override all the pure virtual functions it inherits? [2 Marks]**

**Ans.** The derived class also remains an abstract class - it too cannot be instantiated - because it still has at least one pure virtual function without a concrete implementation. Only once every pure virtual function has been overridden does a class in the hierarchy become concrete.

**Q16. Differentiate briefly between a friend function and a normal member function. [2 Marks]**

**Ans.** A member function belongs to the class, is called using an object with the dot operator, and has an implicit 'this' pointer to that object. A friend function is not a member of the class at all - it is an ordinary external function that has simply been granted permission to access the class's private/protected members, and it takes the object(s) as explicit parameters instead of using 'this'.

**Q17. Why is inheritance sometimes misused, and what design relationship should be used instead in such cases? [2 Marks]**

**Ans.** Inheritance is misused when classes are related only for convenience or code sharing, not by a genuine IS-A relationship (for example, making Wallet inherit from Person just to reuse a name field). In such cases containership (HAS-A) is the more appropriate design choice, since it models 'possesses' rather than 'is a specialised type of'.

**Q18. What is the significance of the 'override' keyword when redefining a base class's virtual function? [2 Marks]**

**Ans.** The override keyword (introduced in C++11) tells the compiler that a derived class function is intended to override a base class virtual function, and the compiler will raise an error if no matching virtual function actually exists in the base class - this helps catch signature mismatches (like typos) that would otherwise silently create an unrelated new function instead of overriding.

## 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. This set covers additional angles not included in the first broad-answer set.

**Q1. Explain, with a complete table, how each of the three inheritance modes (public, protected, private) changes the accessibility of public, protected, and private base class members in the derived class. [8 Marks]**

The mode of inheritance used when deriving a class controls how the base class's public and protected members appear inside the derived class. Private members of the base class are never directly accessible in the derived class, regardless of the inheritance mode used.

| Base Member \ Inheritance Mode | public inheritance      | protected inheritance   | private inheritance     |
|--------------------------------|-------------------------|-------------------------|-------------------------|
| public member                  | Remains public          | Becomes protected       | Becomes private         |
| protected member               | Remains protected       | Becomes protected       | Becomes private         |
| private member                 | Not accessible directly | Not accessible directly | Not accessible directly |

public inheritance is used when the derived class should preserve an IS-A relationship visible to the outside world (the most common case). protected inheritance is used when further derived classes still need access to the base's public interface, but that interface should not be exposed to ordinary outside code. private inheritance is used purely for implementation reuse, without exposing any IS-A relationship at all - outside code cannot treat the derived object as a base object in this case.

**Q2. Explain runtime polymorphism in C++ and how it depends on inheritance and virtual functions. Illustrate with a program. [8 Marks]**

Runtime polymorphism is the ability of a program to decide, while it is actually running, which version of an overridden function to execute, based on the actual type of object a base class pointer or reference is pointing to. It depends on two features working together: inheritance, which lets several derived classes share a common base type, and virtual functions, which tell the compiler to defer the choice of function to runtime instead of deciding at compile time.

```
class Animal
{
public:
    virtual void sound() { cout << "Some animal sound"; }
};

class Cat : public Animal
{
public:
    void sound() override { cout << "Meow"; }
};

int main()
{
```

```
Animal* a = new Cat();  
a->sound();    // prints "Meow" - resolved at runtime, not compile time  
}
```

Without the virtual keyword, the call `a->sound()` would be resolved at compile time based on the pointer's declared type (`Animal`), always calling `Animal::sound()`. Declaring `sound()` virtual forces the program to check, at runtime, the actual object being pointed to (a `Cat` here), and call `Cat::sound()` instead - this mechanism underlies flexible designs such as processing an array of different shape or employee types through a single base pointer/loop.

**Q3. Explain the two distinct problems that can occur with multiple inheritance - plain ambiguity and the diamond problem - and how their solutions differ. [8 Marks]**

Multiple inheritance can cause two related but distinct issues. The first is plain ambiguity: when two unrelated base classes both define a member with the same name, the derived class does not know which one is meant. The second is the diamond problem: a special case where two base classes are themselves derived from one common ancestor, causing that ancestor's members to be duplicated inside the final derived class.

- Plain ambiguity (two unrelated classes with same-named members) is solved using the scope resolution operator to specify the intended base class, or by redefining the member in the derived class itself.
- The diamond problem (duplicate copies of a common ancestor) is solved differently - not by qualifying which copy to use, but by preventing the duplication in the first place, using virtual base classes so that only one shared copy of the common ancestor ever exists.

It is important to distinguish the two: applying the scope resolution operator does not fix a diamond problem (there would still be two physical copies of data, just accessed explicitly), and declaring a virtual base class does not, by itself, solve an unrelated same-name ambiguity between two independent classes.

**Q4. Explain the order of constructor and destructor execution in a hierarchy that combines multiple inheritance with a virtual base class. [8 Marks]**

When a hierarchy combines ordinary multiple inheritance with a virtual base class, C++ follows a specific, well-defined order:

- The virtual base class's constructor is called first of all, and it is called directly by the most-derived class - not by the intermediate classes, even if they also mention it in their own initializer lists.
- Next, the remaining (non-virtual) base classes' constructors run, in the order they are listed in the derived class's own base list.
- Then the derived class's own constructor body runs.
- Destruction happens in exactly the reverse order: derived class destructor first, then the non-virtual base destructors (in reverse of their construction order), and finally the virtual base class destructor last.

This ordering guarantees that the single shared virtual base sub-object is always ready before any class that depends on it is constructed, and is torn down only after every class that might still reference it has already been destroyed.

**Q5. Compare inheritance and containership as two approaches to code reuse, and explain when each should be preferred. [8 Marks]**

Both inheritance and containership let one class reuse the work already done in another class, but they do so through fundamentally different relationships and should be chosen based on what the classes actually represent, not merely for convenience.

| Aspect                | Inheritance (IS-A)                                                                           | Containership (HAS-A)                                                                          |
|-----------------------|----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| Relationship modelled | Derived class is a specialised kind of base class                                            | Class owns/contains an object of another class as a part                                       |
| Coupling              | Tighter - derived class exposes/inherits base's interface                                    | Looser - outer class controls exactly what of the inner object is exposed                      |
| When to prefer        | When the derived type should be usable anywhere the base type is expected (substitutability) | When one object is simply a component or attribute of another, with no substitutability needed |
| Example               | Savings Account IS-A Bank Account                                                            | Car HAS-A Engine, Library HAS-A Books                                                          |

A widely used design guideline is to prefer containership over inheritance unless a genuine, provable IS-A relationship exists, because containership keeps classes more loosely coupled and easier to change independently, while inheritance should be reserved for cases where the derived class must be truly substitutable for the base class.

**Q6. Explain namespaces in detail, including the difference between a using-declaration and a using-directive, with examples. [8 Marks]**

A namespace groups related identifiers (classes, functions, variables) under one name so they do not clash with identically named identifiers defined elsewhere. Members of a namespace are normally accessed with the scope resolution operator, e.g., `NamespaceName::member`. C++ provides two shorthand ways to avoid repeating that prefix:

```
namespace Shop
{
    int price = 100;
    void bill() { cout << "Billing..."; }
}

int main()
{
    using Shop::price;      // using-declaration: brings in only 'price'
    cout << price;          // OK, no prefix needed
    // bill();              // ERROR: bill() was not declared, still needs Shop::bill()

    using namespace Shop;   // using-directive: brings in everything from Shop
    bill();                 // now OK without prefix
}
```

A using-declaration (`using Shop::price;`) brings only the one named identifier into the current scope, giving finer control and a lower risk of future naming conflicts. A using-directive (`using namespace Shop;`) brings every identifier from that namespace into scope at once - convenient, as seen with `using namespace std;`, but more likely to cause conflicts as a program grows and pulls in multiple namespaces.

**Q7. Discuss the role of friend functions in operator overloading, explaining why some operators require a friend function while member functions suffice for others. [6 Marks]**

When overloading a binary operator as a member function, the left-hand operand is always the implicit object (accessed via 'this'), so a statement like `obj + 5` works fine as a member function because `obj` is naturally the left operand. However, an expression like `5 + obj`, or `cout << obj`, has a left-hand operand (an `int`, or `cout`) that is not an object of the class at all, so it cannot supply the implicit 'this'. In such cases, the operator must be overloaded as a standalone (non-member) function, and it is declared a friend so that it can still directly access the class's private data for the actual computation.

```
class Complex
{
    int real, imag;
public:
    Complex(int r, int i) : real(r), imag(i) {}
    friend ostream& operator<<(ostream& out, Complex c);    // left operand is cout, not
Complex
};

ostream& operator<<(ostream& out, Complex c)
{
    out << c.real << " + " << c.imag << "i";
    return out;
}
```

This is why the insertion operator (`<<`) and extraction operator (`>>`), and any operator whose left operand is not naturally an object of the class, are almost always implemented as friend functions rather than member functions.

**Q8. What is private inheritance? Explain its purpose with an example, and state when it should be preferred over composition (containership). [6 Marks]**

In private inheritance, both the public and protected members of the base class become private members of the derived class. This means the derived class can still use the base class's functionality internally, but it does not expose an IS-A relationship to the outside world - a Derived object cannot be used wherever a Base object is expected, and outside code cannot call inherited functions on a Derived object directly.

```
class Timer
{
public:
    void start() { cout << "Timer started"; }
};

class Stopwatch : private Timer    // reuses Timer's implementation privately
{
public:
    void begin() { start(); }    // exposes only what Stopwatch chooses to
};
```

Private inheritance is essentially 'implemented in terms of' rather than 'is a' - it is chosen over plain containership mainly when the derived class needs access to protected members of the base, or needs to override the base's virtual functions, neither of which a simple embedded (contained) object could do. When neither of these is needed, containership is generally the simpler and more flexible choice, since it keeps the classes even more loosely coupled.