OBJECT ORIENTED PROGRAMMING (C++)

Chapter 4: Inheritance

Study Notes for Engineering Students

4.1 Concepts of Inheritance

What is Inheritance?

Inheritance is one of the four fundamental pillars of Object Oriented Programming (the other three being Encapsulation, Polymorphism and Abstraction). It is the mechanism by which one class (called the derived class or child class) acquires the properties (data members) and behaviours (member functions) of another already existing class (called the base class or parent class). Inheritance promotes code reusability, reduces redundancy, and establishes a natural hierarchical relationship between classes, similar to how classification works in the real world (for example, a 'Car' and a 'Bus' are both types of 'Vehicle').

The class whose properties are inherited is called the Base Class (or Super Class), and the class that inherits those properties is called the Derived Class (or Sub Class). Once a class is derived from a base class, it automatically gets access to the non-private members of the base class and can further add its own new members or override existing behaviour.

Derived Classes

A derived class is declared by specifying the base class after a colon, along with an access specifier that controls how the inherited members are treated in the derived class. The general syntax is:

```
class Base
{
   // members of base class
};

class Derived : access-specifier Base
{
   // members of derived class
};
```

Here, access-specifier can be public, private, or protected, and it determines the visibility of the base class members inside the derived class. A derived class object contains, as a sub-object, all the non-static data members of its base class, in addition to any data members declared in the derived class itself.

Member Declaration - Protected Access Specifier

C++ provides three access specifiers for class members: private, protected, and public. The protected specifier is special because it exists purely to support inheritance.

- private: Members are accessible only within the same class. They are NOT accessible to derived classes.
- protected: Members are accessible within the same class and also within any class directly or indirectly derived from it, but not accessible from outside the class hierarchy (i.e., not from main() or unrelated classes).
- public: Members are accessible from anywhere the object is visible, including derived classes and outside code.

The protected keyword strikes a balance: it hides the member from the outside world (like private) while still allowing derived classes to access and modify it directly (like public). This is essential when a derived class needs to work with inherited data without going through accessor/mutator functions.

Effect of Inheritance Mode on Base Class Members

Base Class Member	public inheritance	protected inheritance	private inheritance
public	public	protected	private
protected	protected	protected	private
private	not inherited (inaccessible)	not inherited (inaccessible)	not inherited (inaccessible)

Note: private members of the base class are never directly accessible in the derived class regardless of the mode of inheritance; they can only be accessed indirectly through public or protected member functions of the base class.

Types of Inheritance

Depending on the number of base and derived classes involved, and how they relate to one another, inheritance can be classified into the following types:

1. Single Inheritance

In single inheritance, a derived class inherits from only one base class. This is the simplest and most common form of inheritance.

```
class Animal { ... };
class Dog : public Animal { ... };   // Dog inherits from Animal only
```

2. Multilevel Inheritance

In multilevel inheritance, a derived class is created from another derived class, forming a chain (grandparent -> parent -> child). A class D1 can be derived from base B, and another class D2 can then be derived from D1, and so on.

```
class A { ... };
class B : public A { ... };   // B inherits from A
class C : public B { ... };   // C inherits from B (and indirectly from A)
```

3. Multiple Inheritance

In multiple inheritance, a single derived class inherits from two or more base classes simultaneously. This allows the derived class to combine features from multiple independent classes.

```
class A { ... };
class B { ... };
class C : public A, public B { ... };   // C inherits from both A and B
```

4. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class. This is useful when several sub-classes share common properties defined once in a common base class.

```
class Vehicle { ... };
class Car : public Vehicle { ... };
class Bike : public Vehicle { ... };
class Truck : public Vehicle { ... };
```

5. Hybrid (Virtual) Inheritance

Hybrid inheritance is a combination of two or more types of inheritance listed above (for example, a mix of hierarchical and multiple inheritance). It is typically used when a program needs to combine several relationships at once, and it is the type of inheritance most likely to lead to the Diamond Problem, which is solved in C++ using virtual base classes (discussed in section 4.2).

Ambiguity in Multiple Inheritance

When a class inherits from two or more base classes that each contain a member function or data member with the same name, the compiler cannot determine which member the derived class object is referring to. This situation is called ambiguity.

```
class A { public: void show() { ... } };
class B { public: void show() { ... } };
class C : public A, public B { };

int main() {
   C obj;
   obj.show();   // ERROR: ambiguous - which show()? A's or B's?
}
```

Ambiguity of this kind can be resolved using the following techniques:

- Scope resolution operator: explicitly qualify which base class's member is intended, e.g., obj.A::show() or obj.B::show().
- Defining the function again in the derived class: the derived class can redefine show(), and this new definition takes precedence over both base class versions.
- Virtual base classes: when the ambiguity arises specifically because two base classes were themselves derived from a common grandparent class (the diamond problem), declaring that common base as virtual ensures only one copy of it is inherited, removing the duplication that caused the ambiguity.

4.2 Virtual Base Classes, Abstract Classes, Constructors in Derived Classes

Virtual Base Classes

A common problem in hybrid/multiple inheritance is the Diamond Problem: if classes B and C are both derived from a common base class A, and a class D inherits from both B and C, then D ends up with two separate copies of A's members - one inherited through B, and another through C. This causes ambiguity whenever D tries to access a member of A.

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

D obj;
obj.x = 10;   // ERROR: ambiguous, which copy of x - via B or via C?
```

This is resolved by declaring the common base class as a virtual base class in the intermediate classes. When A is inherited virtually by both B and C, C++ ensures that only ONE shared copy of A is included in D, no matter how many paths lead to it.

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

D obj;
obj.x = 10;   // OK: only one copy of A exists in D
```

The keyword virtual is placed before or after the access specifier in the base list of the intermediate (B and C) classes. Virtual base classes ensure a single, shared sub-object of the common ancestor is present, which also affects how constructors are called (the most-derived class becomes directly responsible for invoking the virtual base's constructor).

Abstract Classes

An abstract class is a class that is designed only to act as a base class and is never meant to be instantiated on its own. It typically represents a generalized concept, leaving the specific implementation details to its derived classes. In C++, a class becomes abstract when it contains at least one pure virtual function.

A pure virtual function is a virtual function that has no implementation in the base class and is declared by assigning it to 0:

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

```
};

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

Because Shape contains a pure virtual function, an object of type Shape cannot be created directly (Shape s; would be a compile-time error). However, a pointer or reference of type Shape can point to objects of derived classes such as Circle, which is what enables runtime polymorphism. A derived class must override every pure virtual function it inherits in order to become a concrete (instantiable) class itself; otherwise it too remains abstract.

Constructors in Derived Classes

Constructors are not inherited in C++ (prior to C++11's inheriting-constructor feature, which is outside the scope of this chapter), meaning the derived class must define its own constructor if it needs to initialize its own members. However, when a derived class object is created, the base class's constructor is always executed first, followed by the derived class's constructor. This ensures the base part of the object is fully initialized before the derived part is built on top of it.

- Order of execution: Base class constructor runs first -> then Derived class constructor runs.
- Order of destruction: Exactly the reverse - Derived class destructor runs first -> then Base class destructor runs.
- Passing arguments to the base constructor: if the base class constructor takes arguments, the derived class constructor must explicitly pass them using a member initializer list.

```
class Base
{
public:
   Base(int x) { cout << "Base constructor, x = " << x; }
};

class Derived : public Base
{
public:
   Derived(int x, int y) : Base(x)   // explicitly calling base constructor
   {
      cout << "Derived constructor, y = " << y;
   }
};
```

In multiple inheritance, base class constructors are called in the order in which the base classes are listed in the derived class's declaration (not the order in which they appear in the initializer list). In the presence of virtual base classes, the virtual base's constructor is called first, by the most-derived class, before any non-virtual base constructors.

4.3 Class Within Class, Containership, IS-A and HAS-A, Namespaces

Class Within Class (Nested Classes)

C++ allows a class to be declared inside another class. Such a class is known as a nested class, and the class that contains it is called the enclosing class. A nested class is primarily used when the inner class exists purely to support the outer class and has no meaning outside its context.

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

int main() {
   Outer::Inner obj;   // accessed using scope resolution
   obj.display();
}
```

Containership (Composition / HAS-A via Embedding)

Containership, also called composition, refers to the situation where an object of one class is declared as a data member (embedded) inside another class. This is different from a nested class definition - here it is not the class declaration that is nested, but an object (instance) of one class that becomes part of another class's data.

```
class Engine
{
public:
   int horsepower;
};

class Car
{
   Engine e;   // Engine object embedded inside Car - containership
public:
   void setPower(int hp) { e.horsepower = hp; }
};
```

When a Car object is created, its embedded Engine sub-object is constructed first (its constructor runs before the Car's own constructor body executes), and when the Car object is destroyed, the Engine sub-object is destroyed after the Car's own destructor finishes - the same 'first-in, last-out' rule that governs inheritance, applied to member objects instead of base classes.

IS-A and HAS-A Relationships

Both inheritance and containership are means of building relationships between classes, but they represent conceptually different kinds of relationships:

- IS-A relationship: Represented by inheritance. It means the derived class is a specialized kind of the base class. For example, 'A Car IS-A Vehicle' - so class Car : public Vehicle correctly models this relationship.
- HAS-A relationship: Represented by containership/composition. It means one class possesses, contains, or is composed of an object of another class as one of its parts. For example, 'A Car HAS-A Engine' - so embedding an Engine object inside the Car class models this relationship.

Difference between IS-A and HAS-A

Basis	**IS-A (Inheritance)**	**HAS-A (Containership)**
Relationship type	Specialization / generalization	Ownership / part-whole
Mechanism	Derived class inherits from base class	One class contains an object of another as a member
Access to members	Derived class can directly access public/protected base members	Outer class accesses the embedded object's members only through the object (dot operator), respecting its access specifiers
Example	Dog IS-A Animal	Car HAS-A Engine
Reusability style	Behaviour and interface reuse through hierarchy	Reuse by assembling existing classes as building blocks

Namespaces

As programs grow larger and combine code from multiple libraries or modules, the possibility of naming conflicts increases - two different pieces of code might unintentionally use the same name for a class, function, or variable. A namespace is a declarative region that provides a scope for the identifiers (names of classes, functions, variables) inside it, helping to avoid such name collisions.

```
namespace First
{
   int value = 10;
   void display() { cout << "First namespace"; }
}

namespace Second
{
   int value = 20;
   void display() { cout << "Second namespace"; }
}

int main() {
   cout << First::value;    // 10
   cout << Second::value;   // 20
   using namespace First;   // brings First's names into scope
   display();               // calls First::display()
}
```

The scope resolution operator (::) is used to access a specific member of a namespace, as in First::value. The statement using namespace X; brings all names from namespace X into the current scope so they can be used without the prefix - this is exactly how the C++ Standard Library's std namespace is commonly used via using namespace std;. Namespaces can also be nested, and the same namespace name can be reopened in multiple places to add more members to it.

4.4 Friend Function, Friend Class, Advantages and Disadvantages of Friends

Friend Function

Normally, private and protected members of a class can only be accessed by the member functions of that class. A friend function is a function that is NOT a member of a class but is still granted permission to access the class's private and protected members. It is declared inside the class using the friend keyword, but it is defined outside the class like an ordinary function (without the class's scope resolution and without the this pointer, since it is not a member).

```
class Box
{
   int length;
public:
   Box(int l) : length(l) {}
   friend void printLength(Box b);   // friend declaration
};

void printLength(Box b)   // definition - note: no Box:: prefix
{
   cout << "Length: " << b.length;   // can access private member directly
}
```

A friend function can also be a member function of another class, and a single function can be declared as a friend in more than one class simultaneously, allowing it to access the private data of all of them.

Friend Class

Just as an individual function can be made a friend, an entire class can be declared as a friend of another class. When class B is declared as a friend of class A, every member function of class B gets access to the private and protected members of class A.

```
class A
{
   int secret;
   friend class B;   // B is a friend class of A
public:
   A() : secret(42) {}
};

class B
```

```
{
public:
   void reveal(A obj) { cout << obj.secret; }   // allowed - B is a friend of A
};
```

Friendship in C++ is granted, not taken: class A must explicitly declare class B as its friend; B cannot declare itself a friend of A. Friendship is also not symmetric (A befriending B does not automatically make B share its private data with A unless declared separately) and not transitive (a friend of a friend is not automatically a friend), and it is not inherited by derived classes of the friend.

Advantages of Friend Functions/Classes

- They allow selective, controlled access to private data of a class without making that data fully public, thereby preserving encapsulation for all other outside code.
- They are especially useful when a function needs to operate on the private members of two or more different classes simultaneously (for example, an operator overload that combines two unrelated classes).
- They provide flexibility in designing operator overloading functions, particularly binary operators where the left-hand operand is not an object of the class (e.g., overloading << for cout).
- They can improve efficiency by avoiding the overhead of extra public getter/setter function calls when tight, well-justified coupling between two classes is required.

Disadvantages of Friend Functions/Classes

- They violate the principle of data hiding/encapsulation to some extent, since an outside function or class gets direct access to private implementation details.
- Overuse of friends increases coupling between classes - a change in the internal representation of a class may now require changes in every friend function/class that directly accesses those members.
- Friendship cannot be inherited, revoked selectively per-member, or granted conditionally, which reduces flexibility compared to well-designed public interfaces.
- Excessive reliance on friends is often considered poor object-oriented design practice, as it blurs the boundaries between classes that OOP encourages to be independent and self-contained.

Multiple Choice Questions (MCQs)

Answer the following 15 questions. Each question carries one correct option. The answer key is provided at the end of this section.

1. Which of the following best describes the class from which properties are inherited in C++ inheritance?

(A) Derived class (B) Base class (C) Friend class (D) Abstract class

2. Which access specifier allows a member to be accessed by derived classes but not by code outside the class hierarchy?

(A) private (B) public (C) protected (D) friend

3. A class D1 is derived from class B, and another class D2 is derived from D1. This is an example of:

(A) Multiple inheritance (B) Hierarchical inheritance (C) Multilevel inheritance (D) Hybrid inheritance

4. When a single derived class inherits from two or more base classes at the same level, it is called:

(A) Multilevel inheritance (B) Multiple inheritance (C) Single inheritance (D) Hierarchical inheritance

5. Several classes such as Car, Bike, and Truck are all derived directly from a single base class Vehicle. This is an example of:

(A) Multiple inheritance (B) Multilevel inheritance (C) Hierarchical inheritance (D) Hybrid inheritance

6. Ambiguity in multiple inheritance, where two base classes have functions with identical names, can be resolved using:

(A) The new keyword (B) The scope resolution operator (C) The friend keyword (D) The static keyword

7. The 'Diamond Problem' in C++ inheritance arises specifically in:

(A) Single inheritance (B) Multilevel inheritance (C) Hierarchical inheritance (D) Hybrid inheritance involving a common ancestor

8. Which keyword is used to ensure only one copy of a common base class is inherited when it is shared through multiple paths?

(A) static (B) virtual (C) friend (D) const

9. A class becomes an abstract class in C++ when it contains at least one:

(A) Static member function (B) Friend function (C) Pure virtual function (D) Constructor

10. In C++, which of the following statements about constructor execution order in inheritance is correct?

(A) Derived class constructor executes before base class constructor (B) Base class constructor executes before derived class constructor (C) Both execute simultaneously (D) Order is decided at runtime randomly

11. When an object of one class is declared as a data member inside another class, this relationship is known as:

(A) Inheritance (B) Friendship (C) Containership (D) Polymorphism

12. 'A Car IS-A Vehicle' is best modelled in C++ using:

(A) Containership (B) A friend function (C) Inheritance (D) A namespace

13. 'A Car HAS-A Engine' is best modelled in C++ using:

(A) Inheritance (B) Containership (composition) (C) Abstract classes (D) Virtual base classes

14. Which C++ feature helps avoid naming conflicts between identifiers from different libraries or modules?

(A) Namespace (B) Friend class (C) Virtual base class (D) Nested loop

15. Which of the following statements about a friend function is TRUE?

(A) It is a member function and uses the this pointer (B) It can access private and protected members despite not being a class member (C) It must be declared inside every class it wants to access (D) It is automatically inherited by derived classes

Answer Key

#	Answer	#	Answer
1	(B) Base class	9	(C) Pure virtual function
2	(C) protected	10	(B) Base class constructor executes before derived class constructor
3	(C) Multilevel inheritance	11	(C) Containership
4	(B) Multiple inheritance	12	(C) Inheritance
5	(C) Hierarchical inheritance	13	(B) Containership (composition)
6	(B) The scope resolution operator	14	(A) Namespace
7	(D) Hybrid inheritance involving a common ancestor	15	(B) It can access private and protected members despite not being a class member
8	(B) virtual		

Broad / Descriptive Questions

Answer the following questions in detail with suitable examples and diagrams/code snippets wherever applicable.

1. What is inheritance in C++? Explain its significance in Object Oriented Programming with a suitable example.

2. Explain the role of the protected access specifier in inheritance. How does it differ from private and public access specifiers?

3. Describe, with syntax and diagrams, the different types of inheritance supported in C++: single, multilevel, multiple, hierarchical, and hybrid inheritance.

4. What is ambiguity in multiple inheritance? Discuss, with a suitable program, how such ambiguity can arise and how it can be resolved.

5. What is the Diamond Problem in C++? Explain how virtual base classes solve this problem, with a program illustrating the concept.

6. What is an abstract class? Explain the concept of a pure virtual function with an example, and state why an abstract class cannot be instantiated.

7. Explain the order in which constructors and destructors are called in a class hierarchy involving both single inheritance and containership. Support your answer with an example program.

8. Differentiate between the IS-A relationship and the HAS-A relationship in Object Oriented Programming. Illustrate both with real-world examples and corresponding C++ code.

9. What are namespaces in C++? Explain their purpose with a program demonstrating how they help avoid naming conflicts.

10. What is a friend function and a friend class in C++? Explain their use with suitable examples, and discuss their advantages and disadvantages.