3.1 Defining Derived Classes

Unit 3: Inheritance and Polymorphism

Object Oriented Programming with C++

3.1 Defining Derived Classes
Unit 3: Inheritance and Polymorphism
Object Oriented Programming with C++

Lecture 24: Defining Derived Classes

- Unit 3: Inheritance and Polymorphism
- Topic 3.1: Defining Derived Classes
- Instructor: [Your Name]
- Course: Object Oriented Programming with C++ (DI05011021)

Lecture 24: Defining Derived Classes
- Unit 3: Inheritance and Polymorphism
- Topic 3.1: Defining Derived Classes
- Instructor: [Your Name]
- Course: Object Oriented Programming with C++ (DI05011021)

Welcome everyone to Lecture 24. Today marks the beginning of Unit 3, where we dive into one of the most powerful features of Object-Oriented Programming: Inheritance. By the end of this session, you'll understand how to create new classes based on existing ones, a concept that will significantly improve your code's reusability and organization.

Introduction to Inheritance

- Inheritance is a foundational pillar of Object-Oriented Programming (OOP).
- It is the process by which one class acquires the properties (data members) and behaviors (member functions) of another class.
- Analogous to biological inheritance: A child inherits traits from parents.
- Primary Goal: Code Reusability. Write once, reuse multiple times.
- It establishes an 'IS-A' relationship (e.g., A Car IS-A Vehicle).

Introduction to Inheritance
- Inheritance is a foundational pillar of Object-Oriented Programming (OOP).
- It is the process by which one class acquires the properties (data members) and behaviors (member functions) of another class.
- Analogous to biological inheritance: A child inherits traits from parents.
- Primary Goal: Code Reusability. Write once, reuse multiple times.
- It establishes an 'IS-A' relationship (e.g., A Car IS-A Vehicle).

Think about how classification works in the real world. We have general categories like 'Vehicle' and more specific ones like 'Car' or 'Motorcycle'. A Car 'is a' Vehicle. It makes sense that a Car should automatically have all the general properties of a Vehicle (like weight, capacity) without us having to redefine them from scratch. Inheritance allows us to model these hierarchical relationships in code, saving time and reducing errors.

Terminology: Base and Derived Classes

- Base Class (Parent/Super Class): The existing class whose properties and methods are being inherited.
- Derived Class (Child/Sub Class): The new class that inherits properties and methods from the base class.
- A derived class inherits all accessible members of the base class.
- A derived class can also add its own new data members and member functions.
- It can even redefine (override) inherited functions to provide specific implementations.

Terminology: Base and Derived Classes
- Base Class (Parent/Super Class): The existing class whose properties and methods are being inherited.
- Derived Class (Child/Sub Class): The new class that inherits properties and methods from the base class.
- A derived class inherits all accessible members of the base class.
- A derived class can also add its own new data members and member functions.
- It can even redefine (override) inherited functions to provide specific implementations.

It's important to get the terminology right. The class we are inheriting FROM is the Base Class. You might also hear it called the Parent or Super class. The new class we are creating is the Derived Class, or Child or Sub class. The derived class gets everything from the base class (subject to access rules, which we'll cover soon), but it isn't just a carbon copy. It can expand upon the base class by adding new features, making it a specialized version of the base class.

Syntax for Defining a Derived Class

- General Syntax:
- class derived_class_name : visibility_mode base_class_name {
- // members of derived class
- };
- The colon (:) indicates inheritance.
- visibility_mode defines how the base class members are inherited.
- If visibility_mode is omitted, it defaults to 'private' in C++ classes.

Syntax for Defining a Derived Class
- General Syntax:
- class derived_class_name : visibility_mode base_class_name {
- // members of derived class
- };
- The colon (:) indicates inheritance.
- visibility_mode defines how the base class members are inherited.
- If visibility_mode is omitted, it defaults to 'private' in C++ classes.

Here is the exact syntax you need to write in C++. You start with the 'class' keyword, then the name of your new derived class. Then comes a single colon. This colon is the inheritance operator. Next is the visibility mode—which can be public, protected, or private—and finally, the name of the base class you are inheriting from. Inside the braces, you define any additional members specific to the derived class. A common beginner mistake is forgetting the visibility mode; remember that if you omit it, C++ assumes you want private inheritance by default.

Understanding Visibility Modes

- Visibility modes dictate the accessibility of inherited base class members inside the derived class and to outside users.
- There are three types of visibility modes in C++:
- 1. Public Inheritance
- 2. Protected Inheritance
- 3. Private Inheritance
- Crucial Note: The 'private' members of a Base class are NEVER directly accessible in the Derived class, regardless of the visibility mode used.

Understanding Visibility Modes
- Visibility modes dictate the accessibility of inherited base class members inside the derived class and to outside users.
- There are three types of visibility modes in C++:
- 1. Public Inheritance
- 2. Protected Inheritance
- 3. Private Inheritance
- Crucial Note: The 'private' members of a Base class are NEVER directly accessible in the Derived class, regardless of the visibility mode used.

Now we address the 'visibility_mode' part of the syntax. This is a critical concept. When you inherit from a base class, you don't just blindly pull in all its members. The visibility mode acts as a filter, determining the new access level of those inherited members within the derived class. Before we look at each mode, I want to emphasize a golden rule: Private members of a base class are truly private. They are never directly accessible by a derived class. They exist in the derived object, but you must use public or protected base class functions to access them.

Visibility Modes Accessibility Matrix

- Base Class Member Access — Inherited as (Public Mode) — Inherited as (Protected Mode) — Inherited as (Private Mode)
- ———————————————————————————————————
- Private — Not Inherited (Hidden) — Not Inherited (Hidden) — Not Inherited (Hidden)
- Protected — Protected — Protected — Private
- Public — Public — Protected — Private

Visibility Modes Accessibility Matrix
- Base Class Member Access — Inherited as (Public Mode) — Inherited as (Protected Mode) — Inherited as (Private Mode)
- ———————————————————————————————————
- Private — Not Inherited (Hidden) — Not Inherited (Hidden) — Not Inherited (Hidden)
- Protected — Protected — Protected — Private
- Public — Public — Protected — Private

This table is your cheat sheet for inheritance visibility. Let's read it together. Notice the first row: private members of the base class are never inherited in an accessible way. Look at the Public Inheritance column: this is the most common form. It keeps public members public and protected members protected. Now look at Private Inheritance: it takes all public and protected members from the base class and makes them private in the derived class. This means they are completely locked away from the outside world and any further derived classes.

1. Public Inheritance

- Models a true 'IS-A' relationship.
- Public members of the base class become public members of the derived class.
- Protected members of the base class become protected members of the derived class.
- Code Example:
- class Person { public: string name; };
- class Student : public Person {
- public: int rollNo;
- };
- // In main: Student s; s.name = "Alice"; // Valid!

1. Public Inheritance
- Models a true 'IS-A' relationship.
- Public members of the base class become public members of the derived class.
- Protected members of the base class become protected members of the derived class.
- Code Example:
- class Person { public: string name; };
- class Student : public Person {
- public: int rollNo;
- };
- // In main: Student s; s.name = "Alice"; // Valid!

Public inheritance is what you'll use 90% of the time. It perfectly models the IS-A relationship. A Student IS A Person. Because we used public inheritance, the public 'name' property of the Person class remains public in the Student class. Therefore, when we create a Student object 's' in main, we can directly access 's.name'. The interface of the base class is preserved and exposed by the derived class.

2. Protected Inheritance

- Both public and protected members of the base class become protected members of the derived class.
- They are accessible inside the derived class.
- They are accessible in classes derived further from this class.
- They are NOT accessible from outside the class (e.g., in main()).
- Useful when you want to create a deeper hierarchy but hide the base interface from the public.

2. Protected Inheritance
- Both public and protected members of the base class become protected members of the derived class.
- They are accessible inside the derived class.
- They are accessible in classes derived further from this class.
- They are NOT accessible from outside the class (e.g., in main()).
- Useful when you want to create a deeper hierarchy but hide the base interface from the public.

Protected inheritance is less common. If you inherit protectedly, the base class's public and protected members become protected in the derived class. This means the outside world—like your main function—cannot see them anymore. However, if you derive a third class from this derived class, that third class WILL be able to see those members. It's like keeping family secrets within the family line, but not showing them to the general public.

3. Private Inheritance

- Models a 'HAS-A' or 'IMPLEMENTED-IN-TERMS-OF' relationship, rather than 'IS-A'.
- Both public and protected members of the base class become private members of the derived class.
- They are accessible ONLY inside the derived class.
- They are completely hidden from outside the class and from any further derived classes.
- Code Example: class Stack : private Array { ... };

3. Private Inheritance
- Models a 'HAS-A' or 'IMPLEMENTED-IN-TERMS-OF' relationship, rather than 'IS-A'.
- Both public and protected members of the base class become private members of the derived class.
- They are accessible ONLY inside the derived class.
- They are completely hidden from outside the class and from any further derived classes.
- Code Example: class Stack : private Array { ... };

Private inheritance is a very specific tool. It takes all public and protected base members and locks them down as private in the derived class. This cuts off the inheritance chain. If you derive from this class again, the new subclass gets nothing from the original base class. We often use this to implement one class using the machinery of another class, without exposing that machinery. For example, implementing a Stack using a pre-existing Array class. A Stack isn't exactly an Array, but it uses one internally.

Forms of Inheritance

- C++ allows classes to inherit in several architectural structures.
- The five main forms of inheritance are:
- 1. Single Inheritance
- 2. Multiple Inheritance
- 3. Multilevel Inheritance
- 4. Hierarchical Inheritance
- 5. Hybrid Inheritance
- We will examine the architecture and syntax of each.

Forms of Inheritance
- C++ allows classes to inherit in several architectural structures.
- The five main forms of inheritance are:
- 1. Single Inheritance
- 2. Multiple Inheritance
- 3. Multilevel Inheritance
- 4. Hierarchical Inheritance
- 5. Hybrid Inheritance
- We will examine the architecture and syntax of each.

Now that we know how to inherit, let's look at the architectural patterns we can build. Just like you can construct different types of buildings with bricks, you can construct different class hierarchies using inheritance. C++ is extremely flexible here and supports five distinct forms of inheritance. We'll walk through each one, defining its structure and showing a quick example.

1. Single Inheritance

- The simplest form of inheritance.
- A derived class is created from exactly ONE base class.
- Structure: [Base (A)] —¿ [Derived (B)]
- Syntax:
- class A { / ... / };
- class B : public A { / ... / };

1. Single Inheritance
- The simplest form of inheritance.
- A derived class is created from exactly ONE base class.
- Structure: [Base (A)] —¿ [Derived (B)]
- Syntax:
- class A { / ... / };
- class B : public A { / ... / };

Single inheritance is the most straightforward pattern. You have one parent class, and one child class that inherits from it. It's a simple, linear relationship. For instance, a class 'Manager' inheriting from a class 'Employee'. Manager gets all the properties of Employee, and adds things like 'bonus' or 'teamSize'.

2. Multiple Inheritance

- A derived class is created from TWO OR MORE base classes.
- The derived class inherits properties from all its parents.
- Structure: [Base 1 (A)], [Base 2 (B)] —¿ [Derived (C)]
- Syntax:
- class A { / ... / };
- class B { / ... / };
- class C : public A, public B { / ... / };
- Notice the comma-separated list of base classes.

2. Multiple Inheritance
- A derived class is created from TWO OR MORE base classes.
- The derived class inherits properties from all its parents.
- Structure: [Base 1 (A)], [Base 2 (B)] —¿ [Derived (C)]
- Syntax:
- class A { / ... / };
- class B { / ... / };
- class C : public A, public B { / ... / };
- Notice the comma-separated list of base classes.

Multiple inheritance is where C++ distinguishes itself from languages like Java or C#, which do not support it for classes. In C++, a class can have multiple direct parents. For example, a 'SmartPhone' class might inherit from a 'Phone' class and a 'Computer' class simultaneously. You define it by separating the base classes with a comma in the definition, and you must specify a visibility mode for each one.

3. Multilevel Inheritance

- A derived class acts as a base class for another derived class.
- Creates a chain or lineage of inheritance.
- Structure: [Base (A)] —¿ [Derived 1 (B)] —¿ [Derived 2 (C)]
- Syntax:
- class A { / ... / };
- class B : public A { / ... / };
- class C : public B { / ... / };

3. Multilevel Inheritance
- A derived class acts as a base class for another derived class.
- Creates a chain or lineage of inheritance.
- Structure: [Base (A)] —¿ [Derived 1 (B)] —¿ [Derived 2 (C)]
- Syntax:
- class A { / ... / };
- class B : public A { / ... / };
- class C : public B { / ... / };

Multilevel inheritance forms a grandfather-father-child relationship. Class A is the base. Class B inherits from A. Then, Class C inherits from Class B. Through this chain, Class C inherits the features of both B and A. A real-world example: 'Animal' -¿ 'Mammal' -¿ 'Dog'. A Dog gets Mammal features (like fur), and Mammal gets Animal features (like breathing).

4. Hierarchical Inheritance

- ONE base class serves as the parent for MORE THAN ONE derived class.
- Creates a tree-like structure branching down.
- Structure: [Base (A)] —¿ [Derived 1 (B)], [Base (A)] —¿ [Derived 2 (C)]
- Syntax:
- class A { / ... / };
- class B : public A { / ... / };
- class C : public A { / ... / };

4. Hierarchical Inheritance
- ONE base class serves as the parent for MORE THAN ONE derived class.
- Creates a tree-like structure branching down.
- Structure: [Base (A)] —¿ [Derived 1 (B)], [Base (A)] —¿ [Derived 2 (C)]
- Syntax:
- class A { / ... / };
- class B : public A { / ... / };
- class C : public A { / ... / };

Hierarchical inheritance is essentially a tree structure. You start with a single generalized class, and you derive multiple specialized classes from it. Think of a 'Shape' class. From 'Shape', you can derive 'Circle', 'Square', and 'Triangle'. They all share the common properties of a Shape, but branch off in different directions with their own specialized formulas for area or perimeter.

5. Hybrid Inheritance

- A combination of two or more of the previous forms of inheritance.
- Usually a mix of Hierarchical and Multiple Inheritance.
- Example Structure:

lass A

- / \

lass B [Class C]

- \ /

lass D

5. Hybrid Inheritance
- A combination of two or more of the previous forms of inheritance.
- Usually a mix of Hierarchical and Multiple Inheritance.
- Example Structure:
lass A
- / \
lass B [Class C]
- \ /
lass D

Hybrid inheritance is just what it sounds like—a mix-and-match of the other types. The most famous (and notorious) form of hybrid inheritance combines hierarchical and multiple inheritance, creating a diamond shape. Class A is inherited by B and C. Then, Class D inherits from both B and C. This specific structure leads to a well-known issue in C++.

The Diamond Problem in Hybrid Inheritance

- In the Diamond Hybrid pattern (A-¿B, A-¿C, B+C-¿D):
- Class D inherits from B and C.
- Both B and C have their own copy of Class A's members.
- Therefore, Class D receives TWO copies of Class A's members.
- This causes Ambiguity: If D tries to access a member from A, the compiler doesn't know which copy to use (the one from B, or the one from C?).

The Diamond Problem in Hybrid Inheritance
- In the Diamond Hybrid pattern (A-¿B, A-¿C, B+C-¿D):
- Class D inherits from B and C.
- Both B and C have their own copy of Class A's members.
- Therefore, Class D receives TWO copies of Class A's members.
- This causes Ambiguity: If D tries to access a member from A, the compiler doesn't know which copy to use (the one from B, or the one from C?).

Let's look at that diamond shape again. If Class A has a variable called 'data', Class B gets a copy of 'data', and Class C gets a copy of 'data'. Now, Class D inherits from both B and C. This means Class D essentially has two variables called 'data'. If we write 'D.data = 5;', the compiler throws an error. It's ambiguous. It asks, 'Do you mean the data you got from B, or the data you got from C?'. This is called the Diamond Problem.

Solving the Diamond Problem: Virtual Inheritance

- C++ provides 'Virtual Inheritance' to solve this ambiguity.
- When classes B and C inherit virtually from A, only ONE shared copy of A is passed down to D.
- Syntax:
- class B : virtual public A { ... };
- class C : virtual public A { ... };
- class D : public B, public C { ... }; // D now has only one copy of A.

Solving the Diamond Problem: Virtual Inheritance
- C++ provides 'Virtual Inheritance' to solve this ambiguity.
- When classes B and C inherit virtually from A, only ONE shared copy of A is passed down to D.
- Syntax:
- class B : virtual public A { ... };
- class C : virtual public A { ... };
- class D : public B, public C { ... }; // D now has only one copy of A.

To fix the Diamond Problem, we use the 'virtual' keyword during the intermediate inheritance steps. When B and C inherit 'virtually' from A, they are basically telling the compiler, 'If anyone inherits from both of us later, make sure they only get one shared instance of Class A.' This eliminates the duplicate copies in Class D, resolving the ambiguity. We will explore virtual inheritance and virtual functions deeper in upcoming lectures.

Lecture Summary

- Inheritance promotes Code Reusability and models 'IS-A' relationships.
- Syntax requires a colon and a visibility mode (public, protected, private).
- Visibility modes strictly dictate how inherited members can be accessed.
- Private members of a base class are NEVER directly accessible to a derived class.
- We explored 5 forms: Single, Multiple, Multilevel, Hierarchical, and Hybrid.
- Hybrid inheritance can lead to the Diamond Problem, solvable via Virtual Inheritance.

Lecture Summary
- Inheritance promotes Code Reusability and models 'IS-A' relationships.
- Syntax requires a colon and a visibility mode (public, protected, private).
- Visibility modes strictly dictate how inherited members can be accessed.
- Private members of a base class are NEVER directly accessible to a derived class.
- We explored 5 forms: Single, Multiple, Multilevel, Hierarchical, and Hybrid.
- Hybrid inheritance can lead to the Diamond Problem, solvable via Virtual Inheritance.

To wrap up, inheritance is your tool for building logical, hierarchical, and reusable code structures. We've seen how to define derived classes, how to control access using visibility modes, and the various architectural shapes our inheritance trees can take. Make sure you practice writing these syntaxes and specifically experiment with how public vs. private inheritance affects what you can access in your main function.