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 20: Defining Derived Classes

- Unit 3: Inheritance and Polymorphism
- Topic 3.1: Defining Derived Classes
- - The Concept of Inheritance
- - Forms of Inheritance
- - Visibility Modes in C++

Lecture 20: Defining Derived Classes
- Unit 3: Inheritance and Polymorphism
- Topic 3.1: Defining Derived Classes
- - The Concept of Inheritance
- - Forms of Inheritance
- - Visibility Modes in C++

Welcome to Lecture 20. Today we begin Unit 3, focusing on Inheritance and Polymorphism. These are the features that make C++ a truly powerful object-oriented language. We will start by defining what derived classes are, look at the different structural forms inheritance can take, and closely examine visibility modes, which dictate how data and methods are passed down.

What is Inheritance?

- Inheritance is the process of creating a new class (derived class) from an existing class (base class).
- The derived class inherits attributes (data members) and behaviors (member functions) of the base class.
- Primary Goal: Code Reusability. Avoid writing the same code twice.
- Establishes an 'IS-A' relationship.
- Example: A 'Car' IS-A 'Vehicle'.

What is Inheritance?
- Inheritance is the process of creating a new class (derived class) from an existing class (base class).
- The derived class inherits attributes (data members) and behaviors (member functions) of the base class.
- Primary Goal: Code Reusability. Avoid writing the same code twice.
- Establishes an 'IS-A' relationship.
- Example: A 'Car' IS-A 'Vehicle'.

Imagine you have a class called Vehicle with properties like speed and a start engine method. If you want to create a Car class, you don't need to rewrite the speed logic. You can inherit from Vehicle. The Car automatically gets those features, and then you can add Car-specific features like trunk capacity. This 'IS-A' relationship is the hallmark of inheritance and leads to highly reusable, organized code.

Base Class vs. Derived Class

- Base Class (Superclass / Parent Class):
- - The original class whose properties and methods are being inherited.
- Derived Class (Subclass / Child Class):
- - The new class created by inheriting from the base class.
- - Inherits accessible members from the base.
- - Can define its own additional members.
- - Can redefine (override) inherited methods.

Base Class vs. Derived Class
- Base Class (Superclass / Parent Class):
- - The original class whose properties and methods are being inherited.
- Derived Class (Subclass / Child Class):
- - The new class created by inheriting from the base class.
- - Inherits accessible members from the base.
- - Can define its own additional members.
- - Can redefine (override) inherited methods.

Let's align on terminology. The class we are borrowing from is the Base Class, or Parent class. The new class doing the inheriting is the Derived Class, or Child class. It's crucial to understand that a derived class is a more specialized version of the base class. It has everything the base class has (that it's allowed to see), plus its own unique additions.

Syntax for Defining a Derived Class

- class DerivedClassName : visibility_mode BaseClassName {
- // Body of the derived class
- // New data members and member functions
- };
-
- - : indicates inheritance.
- - `visibility_mode`: Can be `public`, `protected`, or `private`.
- - If omitted, the default is `private` for classes.

Syntax for Defining a Derived Class
- class DerivedClassName : visibility_mode BaseClassName {
- // Body of the derived class
- // New data members and member functions
- };
-
- - : indicates inheritance.
- - visibility_mode: Can be public, protected, or private.
- - If omitted, the default is private for classes.

Here is the C++ syntax. We declare the class normally, but before the opening brace, we add a colon. The colon means 'inherits from'. Then we specify a visibility mode, followed by the name of the Base class. A very common beginner mistake is forgetting the visibility mode. If you forget it on a 'class', C++ defaults it to 'private', which usually breaks your code immediately by hiding everything.

Forms of Inheritance

- C++ allows classes to inherit in several different structural patterns depending on the real-world relationships being modeled:
- 1. Single Inheritance
- 2. Multiple Inheritance
- 3. Multilevel Inheritance
- 4. Hierarchical Inheritance
- 5. Hybrid Inheritance

Forms of Inheritance
- C++ allows classes to inherit in several different structural patterns depending on the real-world relationships being modeled:
- 1. Single Inheritance
- 2. Multiple Inheritance
- 3. Multilevel Inheritance
- 4. Hierarchical Inheritance
- 5. Hybrid Inheritance

Class hierarchies aren't always a simple one-to-one relationship. To accurately model complex systems, C++ supports five distinct forms of inheritance. We will examine the structure and use cases for each of these over the next few slides.

1. Single Inheritance

- Definition: A derived class is inherited from only one single base class.
- Structure: A -¿ B
- Relationship: One Parent -¿ One Child.
- Example: Class `Manager` inherits from Class `Employee`.

1. Single Inheritance
- Definition: A derived class is inherited from only one single base class.
- Structure: A -¿ B
- Relationship: One Parent -¿ One Child.
- Example: Class Manager inherits from Class Employee.

Single inheritance is the most straightforward. One base class, one derived class. If you have an Employee class with a salary, and a Manager class that gets a bonus, Manager inherits from Employee. It's simple, clean, and the most frequently used form of inheritance.

Code Example: Single Inheritance

```
class Animal {
public:
void eat() { cout ¡¡ "Eating...\n"; }
};

class Dog : public Animal { // Dog inherits Animal
public:
void bark() { cout ¡¡ "Barking...\n"; }
};

// Usage:
// Dog d;
// d.eat(); // Inherited method
// d.bark(); // Own method
```


Code Example: Single Inheritance
class Animal {
public:
void eat() { cout ¡¡ "Eating...\n"; }
};
class Dog : public Animal { // Dog inherits Animal
public:
void bark() { cout ¡¡ "Barking...\n"; }
};
// Usage:
// Dog d;
// d.eat(); // Inherited method
// d.bark(); // Own method

In this snippet, Dog publicly inherits from Animal. Notice how the Dog object 'd' can call 'eat()'. The compiler knows that because Dog is an Animal, it has access to the public 'eat()' function, even though it isn't explicitly written inside the Dog class block.

2. Multiple Inheritance

- Definition: A single derived class inherits from two or more distinct base classes.
- Structure: A, B -¿ C
- Relationship: Multiple Parents -¿ One Child.
- Example: Class `SmartPhone` inherits from Class `Phone` and Class `Camera`.

2. Multiple Inheritance
- Definition: A single derived class inherits from two or more distinct base classes.
- Structure: A, B -¿ C
- Relationship: Multiple Parents -¿ One Child.
- Example: Class SmartPhone inherits from Class Phone and Class Camera.

Multiple inheritance is a powerful feature in C++ that is not present in all OOP languages (like Java). It allows a class to combine features from multiple distinct, unrelated base classes. A SmartPhone is both a communication device (Phone) and an imaging device (Camera), so it can inherit the features of both.

Code Example: Multiple Inheritance

```
class Printer {
public: void print() { cout ¡¡ "Printing...\n"; }
};
class Scanner {
public: void scan() { cout ¡¡ "Scanning...\n"; }
};

// Inheriting from multiple classes
class Copier : public Printer, public Scanner {
// Can use both print() and scan()
};
```


Code Example: Multiple Inheritance
class Printer {
public: void print() { cout ¡¡ "Printing...\n"; }
};
class Scanner {
public: void scan() { cout ¡¡ "Scanning...\n"; }
};
// Inheriting from multiple classes
class Copier : public Printer, public Scanner {
// Can use both print() and scan()
};

To achieve multiple inheritance, we list the base classes separated by commas. Notice that you MUST specify the visibility mode for each base class individually. Here, Copier publicly inherits from both Printer and Scanner, acquiring the public interfaces of both.

3. Multilevel Inheritance

- Definition: A derived class is created from another derived class.
- Structure: A -¿ B -¿ C
- Relationship: Grandparent -¿ Parent -¿ Child.
- The child class inherits features from all ancestral classes.
- Example: `Vehicle` -¿ `Car` -¿ `SportsCar`.

3. Multilevel Inheritance
- Definition: A derived class is created from another derived class.
- Structure: A -¿ B -¿ C
- Relationship: Grandparent -¿ Parent -¿ Child.
- The child class inherits features from all ancestral classes.
- Example: Vehicle -¿ Car -¿ SportsCar.

Multilevel inheritance creates a lineage. A class acts as a derived class, but simultaneously acts as a base class for the next level down. A SportsCar inherits from Car. But since Car inherits from Vehicle, SportsCar indirectly inherits from Vehicle as well, getting all the traits down the chain.

4. Hierarchical Inheritance

- Definition: Multiple derived classes are created from a single common base class.
- Structure: A -¿ B, A -¿ C, A -¿ D
- Relationship: One Parent -¿ Multiple Children.
- Example: `Shape` is the base class. `Circle`, `Square`, and `Triangle` all inherit from `Shape`.

4. Hierarchical Inheritance
- Definition: Multiple derived classes are created from a single common base class.
- Structure: A -¿ B, A -¿ C, A -¿ D
- Relationship: One Parent -¿ Multiple Children.
- Example: Shape is the base class. Circle, Square, and Triangle all inherit from Shape.

Hierarchical inheritance is characterized by a tree-like structure fanning outwards. You start with a general concept—like a Shape. Then you create many specific versions of that concept—Circle, Square, Triangle—which all inherit the common properties (like an 'area' or 'color' attribute) from the single base Shape class.

5. Hybrid Inheritance

- Definition: A combination of two or more types of inheritance.
- Usually a mix of Hierarchical and Multiple Inheritance.
- Complexity: Can lead to the 'Diamond Problem' where a class ends up inheriting duplicate copies of a base class.
- Example: Class A -¿ B & C. Then B & C -¿ Class D.

5. Hybrid Inheritance
- Definition: A combination of two or more types of inheritance.
- Usually a mix of Hierarchical and Multiple Inheritance.
- Complexity: Can lead to the 'Diamond Problem' where a class ends up inheriting duplicate copies of a base class.
- Example: Class A -¿ B & C. Then B & C -¿ Class D.

Hybrid inheritance is just mixing the previous models. However, it introduces a famous complexity called the Diamond Problem. If D inherits from B and C, and both B and C inherit from A... D gets two separate copies of A's data. This causes ambiguity. C++ resolves this using a concept called 'Virtual Inheritance', which we will cover in a future, more advanced session.

Visibility Modes: The Core Concept

- Visibility modes (access specifiers) answer the question: 'How do the inherited base class members appear within the derived class?'
- Modes: `public`, `protected`, `private`.
- Crucial Rule: `private` members of the Base class are NEVER directly accessible by the Derived class, regardless of the visibility mode used.
- Visibility modes define the ceiling of accessibility for inherited members.

Visibility Modes: The Core Concept
- Visibility modes (access specifiers) answer the question: 'How do the inherited base class members appear within the derived class?'
- Modes: public, protected, private.
- Crucial Rule: private members of the Base class are NEVER directly accessible by the Derived class, regardless of the visibility mode used.
- Visibility modes define the ceiling of accessibility for inherited members.

Now we move to the most critical part of the lecture: Visibility Modes. When you inherit, you decide how open those inherited members will be in the new class. The golden rule you must memorize: Private base members are completely off-limits to the derived class. They are inherited, they exist, but you cannot touch them directly. The visibility mode only affects public and protected base members.

Visibility Mode Transformation Matrix

- — Base Member Access — Inherited via `public` — Inherited via `protected` — Inherited via `private` —
- — :— — :— — :— — :— —
- — `public` — Becomes `public` — Becomes `protected` — Becomes `private` —
- — `protected` — Becomes `protected` — Becomes `protected` — Becomes `private` —
- — `private` — Inaccessible — Inaccessible — Inaccessible —

Visibility Mode Transformation Matrix
- — Base Member Access — Inherited via public — Inherited via protected — Inherited via private —
- — :— — :— — :— — :— —
- — public — Becomes public — Becomes protected — Becomes private —
- — protected — Becomes protected — Becomes protected — Becomes private —
- — private — Inaccessible — Inaccessible — Inaccessible —

This table is the master key to inheritance in C++. If you inherit publicly, access levels remain exactly as they were. If you inherit protectedly, public members are downgraded to protected. If you inherit privately, all accessible members are heavily downgraded to private in the derived class. And look at the bottom row: private base members always remain inaccessible.

1. Public Inheritance

- Syntax: `class Derived : public Base`
- The standard and most widely used form of inheritance.
- Accurately models the 'IS-A' relationship.
- Base `public` members -¿ Derived `public` members.
- Base `protected` members -¿ Derived `protected` members.
- An object of the Derived class can directly access the public members of the Base class.

1. Public Inheritance
- Syntax: class Derived : public Base
- The standard and most widely used form of inheritance.
- Accurately models the 'IS-A' relationship.
- Base public members -¿ Derived public members.
- Base protected members -¿ Derived protected members.
- An object of the Derived class can directly access the public members of the Base class.

Public inheritance is what you will use 95% of the time. It preserves the interface of the base class. If a Vehicle has a public start engine method, and Car publicly inherits Vehicle, a programmer can instantiate a Car object and call start engine on it. The interface is maintained.

2. Protected Inheritance

- Syntax: `class Derived : protected Base`
- Base `public` members -¿ Derived `protected` members.
- Base `protected` members -¿ Derived `protected` members.
- Effect: The public interface of the base class is hidden from the outside world, but remains available to the derived class and its future sub-classes.
- Derived class objects cannot access base public methods directly.

2. Protected Inheritance
- Syntax: class Derived : protected Base
- Base public members -¿ Derived protected members.
- Base protected members -¿ Derived protected members.
- Effect: The public interface of the base class is hidden from the outside world, but remains available to the derived class and its future sub-classes.
- Derived class objects cannot access base public methods directly.

Protected inheritance is less common. It essentially says, 'I want to use the base class's features internally, and I want my subclasses to use them, but I do not want the outside world to know I have them.' The public methods of the base class become protected, meaning you can't call them from main() using a derived object.

3. Private Inheritance

- Syntax: `class Derived : private Base`
- Base `public` members -¿ Derived `private` members.
- Base `protected` members -¿ Derived `private` members.
- Effect: All inherited members are locked down as private. They can be used by the derived class internally, but NOT by further sub-classes.
- Models an 'Implemented-In-Terms-Of' relationship, terminating the inheritance chain for those members.

3. Private Inheritance
- Syntax: class Derived : private Base
- Base public members -¿ Derived private members.
- Base protected members -¿ Derived private members.
- Effect: All inherited members are locked down as private. They can be used by the derived class internally, but NOT by further sub-classes.
- Models an 'Implemented-In-Terms-Of' relationship, terminating the inheritance chain for those members.

Private inheritance is the most restrictive. It takes all accessible base members and makes them private in the derived class. This means if you create a third class inheriting from this derived class, it won't see any of the original base class members. It's often used as an alternative to composition when you specifically need to override virtual functions but don't want to expose an IS-A relationship.

Summary of Lecture 20

- Inheritance creates derived classes from base classes to achieve code reusability.
- The five architectural forms are Single, Multiple, Multilevel, Hierarchical, and Hybrid.
- Private members of a base class are never directly accessible by derived classes.
- Visibility modes (`public`, `protected`, `private`) determine the accessibility ceiling for inherited members.
- `public` inheritance is the standard for representing true hierarchical relationships.

Summary of Lecture 20
- Inheritance creates derived classes from base classes to achieve code reusability.
- The five architectural forms are Single, Multiple, Multilevel, Hierarchical, and Hybrid.
- Private members of a base class are never directly accessible by derived classes.
- Visibility modes (public, protected, private) determine the accessibility ceiling for inherited members.
- public inheritance is the standard for representing true hierarchical relationships.

To wrap up, inheritance is a tool for building class hierarchies and reusing code. The form you choose depends on the logical relationship of your objects. The visibility mode you choose depends on how much encapsulation you want to enforce. In our next session, we will see how constructors fit into this hierarchy.