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 28: Defining Derived Classes

- Unit 3: Inheritance and Polymorphism
- Course: Object Oriented Programming with C++

Lecture 28: Defining Derived Classes
- Unit 3: Inheritance and Polymorphism
- Course: Object Oriented Programming with C++

Welcome, everyone, to Lecture 28. Today we are kicking off Unit 3, which focuses on Inheritance and Polymorphism. These are two of the most powerful pillars of Object-Oriented Programming. We will start by looking at how to define derived classes, understand visibility modes, and explore the different forms of inheritance.

What is Inheritance?

- Mechanism of deriving a new class from an old one.
- The old class is referred to as the 'Base Class' (or Parent Class).
- The new class is referred to as the 'Derived Class' (or Child Class).
- Promotes 'Reusability' of code, saving time and reducing errors.

What is Inheritance?
- Mechanism of deriving a new class from an old one.
- The old class is referred to as the 'Base Class' (or Parent Class).
- The new class is referred to as the 'Derived Class' (or Child Class).
- Promotes 'Reusability' of code, saving time and reducing errors.

Let's start with a formal definition. Inheritance is the process by which objects of one class acquire the properties and behaviors of objects of another class. Think of it like genetic inheritance, where a child inherits traits from their parents. In C++, the existing class is the Base Class, and the newly created class is the Derived Class. The primary motivation for inheritance is code reusability—why write the same code twice when you can just inherit it?

Real-World Analogy of Inheritance

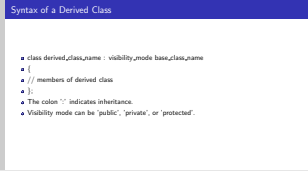
- Base Class: 'Vehicle' (Properties: Wheels, Engine, Color; Methods: Start, Stop)
- Derived Class 1: 'Car' (Inherits Vehicle, adds 'Trunk Size')
- Derived Class 2: 'Motorcycle' (Inherits Vehicle, adds 'Has Sidecar')

Real-World Analogy of Inheritance
- Base Class: 'Vehicle' (Properties: Wheels, Engine, Color; Methods: Start, Stop)
- Derived Class 1: 'Car' (Inherits Vehicle, adds 'Trunk Size')
- Derived Class 2: 'Motorcycle' (Inherits Vehicle, adds 'Has Sidecar')

To visualize this, imagine a general category called 'Vehicle'. All vehicles have certain common properties, like the number of wheels, an engine, and color. They also share behaviors like starting and stopping. Now, if we want to model a 'Car' or a 'Motorcycle', we don't need to redefine what an engine or a wheel is. We simply inherit from 'Vehicle' and add the specific features that make a Car a Car, like trunk size, or a Motorcycle a Motorcycle, like a sidecar.

Syntax of a Derived Class

- class derived_class_name : visibility_mode base_class_name
- {
- // members of derived class
- };
- The colon ':' indicates inheritance.
- Visibility mode can be 'public', 'private', or 'protected'.



Here is the syntax for defining a derived class in C++. We use the 'class' keyword followed by the name of our new derived class. Then, we use a single colon. This colon is crucial—it tells the compiler that inheritance is happening. After the colon, we specify a visibility mode, which dictates how the members of the base class are inherited, followed by the name of the base class. Inside the curly braces, you add the new members specific to the derived class.

Understanding Visibility Modes

- Visibility modes control the access of base class members within the derived class.
- Three modes: Public, Private, Protected.
- Default mode is Private if none is specified.
- Note: Private members of a base class are NEVER directly accessible by the derived class, regardless of visibility mode.

Understanding Visibility Modes
- Visibility modes control the access of base class members within the derived class.
- Three modes: Public, Private, Protected.
- Default mode is Private if none is specified.
- Note: Private members of a base class are NEVER directly accessible by the derived class, regardless of visibility mode.

When inheriting a base class, we must specify how its members will be treated in the derived class. This is done using visibility modes. C++ provides three modes: Public, Private, and Protected. If you forget to write a visibility mode, C++ defaults to Private inheritance for classes. It's an important rule to remember that the private members of a base class remain strictly private to it; they are never directly accessible from the derived class, no matter which visibility mode you choose.

Public Visibility Mode

- Syntax: class Derived : public Base
- Base's public members become public in Derived.
- Base's protected members become protected in Derived.
- Models an 'IS-A' relationship perfectly.

Public Visibility Mode
- Syntax: class Derived : public Base
- Base's public members become public in Derived.
- Base's protected members become protected in Derived.
- Models an 'IS-A' relationship perfectly.

Let's dive into Public inheritance, the most common type. When a class is derived publicly, the public members of the base class remain public in the derived class. They can be accessed by objects of the derived class. The protected members of the base class remain protected in the derived class. This mode perfectly models an 'IS-A' relationship. For example, a Car IS-A Vehicle.

Private Visibility Mode

- Syntax: class Derived : private Base
- Base's public members become private in Derived.
- Base's protected members become private in Derived.
- Used for 'implemented-in-terms-of' rather than 'IS-A'.

Private Visibility Mode
- Syntax: class Derived : private Base
- Base's public members become private in Derived.
- Base's protected members become private in Derived.
- Used for 'implemented-in-terms-of' rather than 'IS-A'.

Next is Private inheritance. When you inherit privately, all the public and protected members of the base class become private members of the derived class. This means they can only be accessed by the member functions of the derived class, and not by any external objects or further derived classes. This is less about an 'IS-A' relationship and more about utilizing the base class's code internally, often called 'implemented-in-terms-of'.

Protected Visibility Mode

- Syntax: class Derived : protected Base
- Base's public members become protected in Derived.
- Base's protected members become protected in Derived.
- Useful when you want to restrict external access but allow further derivation.

Protected Visibility Mode
- Syntax: class Derived : protected Base
- Base's public members become protected in Derived.
- Base's protected members become protected in Derived.
- Useful when you want to restrict external access but allow further derivation.

Finally, Protected inheritance. In this mode, both the public and protected members of the base class become protected members of the derived class. They are hidden from the outside world (like private members) but are still available to any further classes derived from this new class. This is a niche mode, used when you are building deep inheritance hierarchies and need tight control over who can access what.

Summary of Visibility Modes

- Base Member Level — Public Derivation — Private Derivation — Protected Derivation
- Private — Not Inherited — Not Inherited — Not Inherited
- Protected — Protected — Private — Protected
- Public — Public — Private — Protected

Summary of Visibility Modes
- Base Member Level — Public Derivation — Private Derivation — Protected Derivation
- Private — Not Inherited — Not Inherited — Not Inherited
- Protected — Protected — Private — Protected
- Public — Public — Private — Protected

This table is an excellent study guide. Notice the first row: private members of the base class are never inherited, period. Look at the columns to see what happens to protected and public members. Public derivation preserves the original access levels. Private derivation forces everything to private. Protected derivation forces everything to protected. Memorizing this table will save you a lot of debugging time!

Forms of Inheritance Overview

- C++ supports multiple ways to combine classes.
- 1. Single Inheritance
- 2. Multiple Inheritance
- 3. Multilevel Inheritance
- 4. Hierarchical Inheritance
- 5. Hybrid Inheritance

Forms of Inheritance Overview
- C++ supports multiple ways to combine classes.
- 1. Single Inheritance
- 2. Multiple Inheritance
- 3. Multilevel Inheritance
- 4. Hierarchical Inheritance
- 5. Hybrid Inheritance

Now that we know how to inherit, let's look at the architectural patterns of inheritance. Depending on how many base classes or derived classes we have, and how they connect, C++ categorizes inheritance into five main forms: Single, Multiple, Multilevel, Hierarchical, and Hybrid. Let's look at each one with diagrams and examples.

1. Single Inheritance

- Definition: A derived class with only ONE base class.
- Diagram Concept: A -¿ B (A is base, B is derived).
- Example: class Employee inherits from class Person.

1. Single Inheritance
- Definition: A derived class with only ONE base class.
- Diagram Concept: A -¿ B (A is base, B is derived).
- Example: class Employee inherits from class Person.

Single inheritance is the simplest and most common form. You have exactly one base class, and you derive exactly one class from it. Visually, this is just a straight line from Class A down to Class B. For instance, if you have a base class 'Person', you can create a single derived class 'Employee' that inherits the name and age properties from 'Person'.

2. Multilevel Inheritance

- Definition: A class is derived from another derived class.
- Diagram Concept: A -¿ B -¿ C.
- Example: Animal -¿ Mammal -¿ Dog.
- Class B is an intermediate base class.

2. Multilevel Inheritance
- Definition: A class is derived from another derived class.
- Diagram Concept: A -¿ B -¿ C.
- Example: Animal -¿ Mammal -¿ Dog.
- Class B is an intermediate base class.

Multilevel inheritance takes single inheritance and stacks it. Class A is inherited by Class B, and then Class B is inherited by Class C. Here, Class B serves as both a derived class (from A) and a base class (for C). A good biological example is tracing lineage: An Animal class is inherited by a Mammal class, which is then inherited by a Dog class. The Dog inherits traits from both Mammal and Animal.

3. Multiple Inheritance

- Definition: A derived class has MORE THAN ONE base class.
- Diagram Concept: A and B -¿ C.
- Syntax: class C : public A, public B { ... };
- Note: Can lead to the 'Diamond Problem' (Ambiguity).

3. Multiple Inheritance
- Definition: A derived class has MORE THAN ONE base class.
- Diagram Concept: A and B -¿ C.
- Syntax: class C : public A, public B { ... };
- Note: Can lead to the 'Diamond Problem' (Ambiguity).

Multiple inheritance is where C++ shows its power, allowing a single derived class to inherit from two or more distinct base classes simultaneously. Imagine Class A and Class B both pointing down to Class C. For instance, a 'SmartPhone' class could inherit from both a 'Camera' class and a 'Phone' class. Be warned, though: if both base classes have a function with the same name, it creates an ambiguity known as the Diamond Problem, which we will address in later lectures.

4. Hierarchical Inheritance

- Definition: Multiple derived classes inherit from a SINGLE base class.
- Diagram Concept: A -¿ B, A -¿ C, A -¿ D.
- Example: Shape -¿ (Circle, Rectangle, Triangle).

4. Hierarchical Inheritance
- Definition: Multiple derived classes inherit from a SINGLE base class.
- Diagram Concept: A -¿ B, A -¿ C, A -¿ D.
- Example: Shape -¿ (Circle, Rectangle, Triangle).

Hierarchical inheritance is like a tree structure. You start with one general base class at the top, and you derive multiple specific classes from it. For example, a base class 'Shape' can be inherited by 'Circle', 'Rectangle', and 'Triangle'. Each derived class shares the common properties of 'Shape' but implements its own specific area calculations and drawing methods.

5. Hybrid Inheritance

- Definition: A combination of two or more forms of inheritance.
- Often a mix of Multilevel and Multiple inheritance.
- Requires careful design to avoid complexity and ambiguity.
- Virtual base classes are often used to solve ambiguity here.

5. Hybrid Inheritance
- Definition: A combination of two or more forms of inheritance.
- Often a mix of Multilevel and Multiple inheritance.
- Requires careful design to avoid complexity and ambiguity.
- Virtual base classes are often used to solve ambiguity here.

Finally, Hybrid inheritance. This is exactly what it sounds like—a mix-and-match of the other forms. Usually, it's a combination of hierarchical and multiple inheritance. While powerful, it can quickly make your code architecture complex and difficult to maintain. It is the primary scenario where the 'Diamond Problem' occurs, requiring the use of 'virtual base classes' to ensure only one copy of the base class is inherited.

Code Example: Single Public Inheritance

```
class Animal {
public:
void eat() { cout ¡¡ "Eating..."; }
};
class Dog : public Animal {
public:
void bark() { cout ¡¡ "Barking..."; }
};
// In main: Dog d; d.eat(); d.bark();
```


Code Example: Single Public Inheritance
class Animal {
public:
void eat() { cout ¡¡ "Eating..."; }
};
class Dog : public Animal {
public:
void bark() { cout ¡¡ "Barking..."; }
};
// In main: Dog d; d.eat(); d.bark();

Let's look at a quick code snippet to solidify this. We have a base class 'Animal' with a public method 'eat'. We then define a derived class 'Dog' that inherits publicly from 'Animal'. The 'Dog' class defines its own method, 'bark'. In our main function, we can create a Dog object. Because of public inheritance, the Dog object can call 'eat()' from the Animal class, and 'bark()' from its own class.

Lecture Summary

- Inheritance enables code reusability via Base and Derived classes.
- Visibility modes (Public, Private, Protected) dictate access levels.
- Private members of a base class are NEVER inherited.
- Five forms: Single, Multilevel, Multiple, Hierarchical, Hybrid.

Lecture Summary
- Inheritance enables code reusability via Base and Derived classes.
- Visibility modes (Public, Private, Protected) dictate access levels.
- Private members of a base class are NEVER inherited.
- Five forms: Single, Multilevel, Multiple, Hierarchical, Hybrid.

To wrap up today's lecture: We've established that inheritance is the cornerstone of code reusability in C++. We learned how to define a derived class using the colon syntax. We explored how the three visibility modes—Public, Private, and Protected—alter the access of inherited members. And lastly, we surveyed the five topological forms of inheritance. Make sure you practice defining these structures!

What's Next?

- Deep dive into Multiple Inheritance and Ambiguity.
- Understanding the 'Diamond Problem'.
- Introduction to Virtual Base Classes.
- Order of Constructor/Destructor execution in inheritance.

What's Next?
- Deep dive into Multiple Inheritance and Ambiguity.
- Understanding the 'Diamond Problem'.
- Introduction to Virtual Base Classes.
- Order of Constructor/Destructor execution in inheritance.

In our next lecture, we are going to look closer at Multiple Inheritance. We will tackle the dreaded Diamond Problem head-on and learn how Virtual Base Classes solve it. We'll also investigate the hidden mechanics of inheritance: in what order are constructors and destructors called when you create a derived object? Thank you, and I will see you in the next class.