Object Oriented Programming (Spring2024)

Week04 (4/5/6-Mar-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Inheritance

Definition: Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (called the derived or subclass) to inherit the characteristics and behaviors of an existing class (called the base or superclass). The derived class can reuse and extend the functionalities of the base class, creating a hierarchy of classes.

Why is Inheritance Needed?

1. **Code Reusability:** Inheritance facilitates code reuse by allowing a new class to inherit attributes and behaviors from an existing class. This reduces redundancy in code and promotes a more modular and maintainable codebase.
2. **Hierarchy and Organization:** Inheritance allows the creation of class hierarchies, reflecting real-world relationships. It provides a natural way to model entities that share common characteristics.
3. **Efficiency:** Inheritance can lead to more efficient code development. When common functionalities are defined in a base class, changes made to the base class automatically reflect in all derived classes, promoting consistency.
4. **Polymorphism:** Inheritance supports polymorphism, where objects of the derived class can be treated as objects of the base class. This enables flexibility in designing and interacting with objects.

What if We Don't Use Inheritance?

1. **Code Duplication:** Without inheritance, similar functionalities must be implemented separately in different classes, leading to code duplication. This makes the codebase larger, harder to maintain, and increases the likelihood of errors.
2. **Lack of Modularity:** Inheritance promotes modularity by organizing code into reusable components. Without it, code may become monolithic, making it challenging to understand and update.
3. **Limited Flexibility:** Inheritance allows for flexibility in adapting and extending existing code. Without it, making changes or introducing new features might require extensive modifications across multiple classes.
4. **Reduced Readability:** Inheritance provides a clear structure to the relationships between classes. Without it, the relationships may become less apparent, leading to reduced code readability.

Real-World Examples:

1. **Vehicle Hierarchy:**
    - *With Inheritance:* A base class "Vehicle" may have common properties like **speed** and **fuel efficiency**. Derived classes like "Car," "Motorcycle," and "Truck" (e.g., **bed**, **ground clearence**) inherit these properties but can have additional features specific to each type.
    - *Without Inheritance:* Each vehicle type would need to independently implement speed and fuel efficiency, resulting in redundant code.
2. **Employee Management:**
    - *With Inheritance:* A base class "Employee" may have properties like **name**, **ID**, and **salary**. Derived classes like "Manager" (e.g., **team**) and "Engineer" inherit these properties but can have specialized methods or additional attributes.
    - *Without Inheritance:* Each employee type would need to independently implement common properties, leading to code duplication.
3. **Animal Classification:**
    - *With Inheritance:* A base class "Animal" may have common characteristics like **species** and **habitat**. Derived classes like "Mammal," "Reptile," and "Bird" inherit these characteristics but can have specific behaviors.
    - *Without Inheritance:* Each type of animal would need to independently implement common characteristics, making the code less organized.

In short, inheritance is a powerful tool in OOP that promotes code reuse, modularity, and flexibility. It enables the creation of well-organized class hierarchies, reflecting real-world relationships and improving the overall design and maintainability of software systems.

Basic Syntax

In C++, inheritance is a mechanism that allows a new class (the derived or subclass) to inherit properties and behaviors from an existing class (the base or superclass). The basic syntax of inheritance in C++ involves using the `class` keyword along with access specifiers to declare the relationship between the base and derived classes.

Here is the basic syntax of inheritance in C++:

```cpp
// Base class declaration
class BaseClass {
    // Base class members and methods
};

// Derived class declaration inheriting from BaseClass
class DerivedClass : access_specifier BaseClass {
    // Derived class members and methods
};
```

- **Base Class:** It contains the common properties and behaviors that can be shared among multiple classes. The base class serves as a blueprint for derived classes.

- **Derived Class:** It inherits from the base class and can extend or override its functionalities. The derived class inherits the members (data members and member functions) of the base class.
- **Access Specifiers:** The access specifier (`public` , `private` , or `protected`) determines the visibility of the base class members in the derived class. The choice of access specifier affects how the derived class can access the inherited members.

Types of Inheritance

Let's go through each type of inheritance and illustrate the syntax with examples:

1. Single Inheritance: In single inheritance, a class inherits from only one base class.

```
// Base class
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};



// Derived class inheriting publicly from Animal
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();  // Inherited from Animal
    myDog.bark(); // Defined in Dog class
    return 0;
}
```

2. Multiple Inheritance: In multiple inheritance, a class can inherit from more than one base class. This allows the derived class to inherit properties and behaviors from multiple sources.

```cpp
// Base classes
class Shape {
public:
    void draw() {
        cout << "Drawing a shape" << endl;
    }
};


// Derived class inheriting publicly from Shape and Color
class ColoredShape : public Shape, public Color {
public:
    void display() {
        cout << "Displaying a colored shape" << endl;
    }
};

int main() {
    ColoredShape myColoredShape;
    myColoredShape.draw();      // Inherited from Shape
    myColoredShape.setColor("Blue"); // Inherited from Color
    myColoredShape.display();   // Defined in ColoredShape class
    return 0;
}
```

3. Multilevel Inheritance: In multilevel inheritance, a class derives from another class, and then another class derives from that class. It creates a chain of inheritance.

```cpp
// Base class
class Vehicle {
public:
    void start() {
        cout << "Vehicle is starting" << endl;
    }
};

// Intermediate class inheriting from Vehicle
class Car : public Vehicle {
public:
    void accelerate() {
        cout << "Car is accelerating" << endl;
    }
};
```

```cpp
// Derived class inheriting from Car
class SportsCar : public Car {
public:
    void turboBoost() {
        cout << "SportsCar is turbo boosting" << endl;
    }
};

int main() {
    SportsCar mySportsCar;
    mySportsCar.start();       // Inherited from Vehicle
    mySportsCar.accelerate();  // Inherited from Car
    mySportsCar.turboBoost();  // Defined in SportsCar class
    return 0;
}
```

4. Hierarchical Inheritance: In hierarchical inheritance, multiple classes derive from a common base class. It forms a tree-like structure.

Example:

```cpp
#include <iostream>

// Base class
class Vehicle {
public:
    void start() {
        std::cout << "Vehicle is starting" << std::endl;
    }
};
```

```
// Derived class inheriting from Vehicle
class Car : public Vehicle {
public:
    void accelerate() {
        std::cout << "Car is accelerating" << std::endl;
    }
};

// Another derived class inheriting from Vehicle
class Motorcycle : public Vehicle {
public:
    void wheelie() {
        std::cout << "Motorcycle is doing a wheelie" << std::endl;
    }
};



int main() {
    Car myCar;
    myCar.start();      // Inherited from Vehicle
    myCar.accelerate(); // Defined in Car class

    Motorcycle myMotorcycle;
    myMotorcycle.start();   // Inherited from Vehicle
    myMotorcycle.wheelie(); // Defined in Motorcycle class

    return 0;
}
```

These examples illustrate the different types of inheritance in C++, showcasing how classes can be organized and share functionalities through various inheritance mechanisms.

Types of Inheritance: Pros and Cons

Let's discuss the advantages and potential issues associated with each type of inheritance in C++:

1. Single Inheritance:

Advantages:

- **Simplicity:** Single inheritance provides a clear and straightforward hierarchy, making the code more readable.
- **Ease of Maintenance:** With a linear class hierarchy, understanding and maintaining code is generally simpler.

Potential Issues:

- **Limited Reusability:** The derived class can inherit from only one base class, potentially limiting code reuse.
- **Rigidity:** Changes to the base class can impact all derived classes, leading to a more rigid design.

2. Multiple Inheritance:

Advantages:

- **Enhanced Reusability:** Multiple inheritance allows a class to inherit from multiple base classes, promoting greater code reuse.
- **Flexibility:** It provides flexibility in combining functionalities from different sources.

Potential Issues:

- **Ambiguity:** Ambiguities can arise if there are naming conflicts or if the compiler is unsure about which base class a particular member belongs to.
- **Complexity:** Multiple inheritance can lead to complex class hierarchies, making the code harder to understand and maintain.

3. Multilevel Inheritance:

Advantages:

- **Gradual Complexity:** The complexity increases gradually through the hierarchy, making it easier to manage.
- **Hierarchical Structure:** It establishes a clear hierarchical structure, which aids in understanding the relationships between classes.

Potential Issues:

- **Indirect Access:** Indirect access to base class members can make the code less intuitive.
- **Inherited Complexity:** Issues in the base class can propagate to derived classes.

4. Hierarchical Inheritance:

Advantages:

- **Clear Hierarchy:** Hierarchical inheritance provides a clear and intuitive class hierarchy.
- **Code Reusability:** It promotes code reuse among related classes.

Potential Issues:

- **Duplication of Code:** If common functionalities are present in multiple branches of the hierarchy, there may be code duplication.
- **Rigidity:** Changes to the base class can affect multiple branches of the hierarchy.

Base Class-Derived Class Relationship

The relationship between a base class and a derived class is a fundamental concept in object-oriented programming (OOP). This relationship is characterized by inheritance, where a derived class inherits properties and behaviors from a base class. Let's explore this relationship in-depth:

Key Aspects of the Relationship:

1. Inheritance:

- **Definition:** Inheritance is the mechanism through which a derived class inherits the members (attributes and methods) of a base class.
- **Syntax:** `class DerivedClass : access_specifier BaseClass {...};`

- **Example:**

```cpp
// Base class
class Animal {
public:
    void eat() {
        std::cout << "Animal is eating" << std::endl;
    }
};

// Derived class inheriting publicly from Animal
class Dog : public Animal {
public:
    void bark() {
        std::cout << "Dog is barking" << std::endl;
    }
};
```

2. Access Specifiers:

- **Public, Protected, and Private Inheritance:**
    - **Public Inheritance:** Members of the base class remain accessible as public members in the derived class.
    - **Protected Inheritance:** Members of the base class become protected members in the derived class.
    - **Private Inheritance:** Members of the base class become private members in the derived class.
- **Example:**

```cpp
// Public inheritance
class DerivedClass : public BaseClass {...};

// Protected inheritance
class DerivedClass : protected BaseClass {...};
```

3. Base Class Members in Derived Class:

- **Inherited Members:** The derived class inherits all the public and protected members of the base class.
- **Accessibility:** The access specifiers determine the visibility of the inherited members in the derived class.
- **Example:**

```cpp
class BaseClass {
public:
    int publicMember;
protected:
    int protectedMember;
private:
    int privateMember;
};

class DerivedClass : public BaseClass {
public:
    // publicMember is accessible as a public member in DerivedClas
s
    // protectedMember is accessible as a protected member in Deriv
edClass
    // privateMember is not directly accessible in DerivedClass
};
```

4. Method Overriding:

- **Definition:** Derived classes can provide a specific implementation for a method that is already defined in the base class. This is known as method overriding.
- **Syntax:** Use the `override` keyword in the derived class.
- **Example:**

```
// Base class
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a generic shape" << std::endl;
    }
```

5. Constructor Inheritance:

- **Initialization of Base Class:** The derived class constructor initializes the base class part of the object using the base class constructor.
- **Example:**

```
// Base class
class BaseClass {
public:
    BaseClass(int value) {
        // Base class constructor implementation
    }
};

// Derived class
class DerivedClass : public BaseClass {
public:
    DerivedClass(int value, int derivedValue) : BaseClass(value) {
        // Derived class constructor implementation
    }
};
```

6. Destructors in Inheritance:

- **Execution Order:** Destructors are called in the reverse order of constructors. The derived class destructor is called before the base class destructor.
- **Example:**

```cpp
// Base class
class BaseClass {
public:
```

Illustrative Example:

Let's consolidate the concepts into a comprehensive example:

```cpp
#include <iostream>

// Base class
class Animal {
public:
    void eat() {
        std::cout << "Animal is eating" << std::endl;
    }
};

// Derived class inheriting publicly from Animal
class Dog : public Animal {
public:
    void bark() {
        std::cout << "Dog is barking" << std::endl;
    }



    // Overriding the eat method from the base class
    void eat() override {
        cout::cout << "Dog is eating bones" << std::endl;
    }
};

int main() {
    // Creating an object of the derived class
    Dog myDog;

    // Accessing inherited method from the base class
    myDog.eat();

    // Accessing method defined in the derived class
    myDog.bark();

    return 0;
}
```

In this example, `Dog` is a derived class that inherits from the `Animal` base class. It demonstrates the relationship between the base and derived class, showcasing method overriding and inheritance of members.

Inheritance of Data Members

When a derived class inherits from a base class in C++, it inherits both data members and member functions of the base class. The access specifiers (`public` , `protected` , and `private`) play a crucial role in determining the visibility and accessibility of these inherited members. Let's delve into the inheritance of members, with a particular focus on the `protected` access specifier.

1. Public Inheritance:

In public inheritance, public and protected data members of the base class become public and protected members, respectively, in the derived class.

```cpp
class BaseClass {
public:
    int publicData;
protected:
    int protectedData;
private:
    int privateData;
};

class DerivedClass : public BaseClass {
public:
    // publicData is accessible as a public member
    // protectedData is accessible as a protected member
    // privateData is not directly accessible
};
```

2. Protected Inheritance:

In protected inheritance, public and protected data members of the base class become protected members in the derived class.

```
class BaseClass {
public:
    int publicData;
protected:
```

3. Private Inheritance:

In private inheritance, both public and protected data members of the base class become private members in the derived class.

```
class BaseClass {
public:
    int publicData;
protected:
    int protectedData;
private:
    int privateData;
};

class DerivedClass : private BaseClass {
private:
    // publicData is accessible as a private member
    // protectedData is accessible as a private member
    // privateData is not directly accessible
};
```

Inheritance of Member Functions

1. Public Inheritance:

In public inheritance, public and protected member functions of the base class become public and protected member functions, respectively, in the derived class.

```
class BaseClass {
public:
    void publicFunction() {}
protected:
    void protectedFunction() {}
private:
    void privateFunction() {}
};
```

```cpp
class DerivedClass : public BaseClass {
public:
    // publicFunction is accessible as a public member function
    // protectedFunction is accessible as a protected member function
    // privateFunction is not directly accessible
};
```

2. Protected Inheritance:

In protected inheritance, public and protected member functions of the base class become protected member functions in the derived class.

```cpp
class BaseClass {
public:
    void publicFunction() {}
protected:
    void protectedFunction() {}
private:
    void privateFunction() {}
};

class DerivedClass : protected BaseClass {
protected:
    // publicFunction is accessible as a protected member function
    // protectedFunction is accessible as a protected member function
    // privateFunction is not directly accessible
};
```

3. Private Inheritance:

In private inheritance, both public and protected member functions of the base class become private member functions in the derived class.

```
class BaseClass {
public:
    void publicFunction() {}
```

Protected Access Specifier:

The `protected` access specifier is significant when considering inheritance. It allows the derived class to access the protected members of the base class, but these members are not accessible outside the class hierarchy.

- **Protected Data Members:**

  ```
  class BaseClass {
  protected:
      int protectedData;
  };

  class DerivedClass : public BaseClass {
  public:
      void accessProtectedData() {
          // Accessing protectedData from the base class
          protectedData = 10;
      }
  };
  ```

- **Protected Member Functions:**

  ```
  class BaseClass {
  protected:
      void protectedFunction() {}
  };

  class DerivedClass : public BaseClass {
  public:
      void callProtectedFunction() {
          // Calling protectedFunction from the base class
          protectedFunction();
      }
  };
  ```

Method Overriding

Method overriding is a concept in object-oriented programming (OOP) that allows a derived class to provide a specific implementation for a method that is already defined in its base class. The overridden method in the derived class should have the same signature (name and parameters) as the method in the base class. This enables polymorphism, allowing objects of the derived class to be treated as objects of the base class.

Demonstration of Method Overriding:

Let's consider a scenario where we have a base class `Shape` with a method `draw()` and a derived class `Circle` that overrides the `draw()` method to provide a specific implementation for drawing a circle.

```
#include <iostream>

// Base class
class Shape {
public:
    // Virtual function for drawing
    virtual void draw() {
        std::cout << "Drawing a generic shape" << std::endl;
    }
};

// Derived class (Circle) inheriting publicly from Shape
class Circle : public Shape {
public:
    // Overridden draw method for drawing a circle
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }
};



int main() {
    // Creating objects of base and derived classes
    Shape genericShape;
    Circle myCircle;

    // Using objects through base class pointers
    Shape* shapePtr1 = &genericShape;
    Shape* shapePtr2 = &myCircle;

    // Calling draw method on base class pointer
    shapePtr1->draw();  // Output: Drawing a generic shape
    shapePtr2->draw();  // Output: Drawing a circle

    return 0;
}
```

In this example:

1. The `Shape` class has a virtual method `draw()` , which serves as a base for drawing various shapes.
2. The `Circle` class is derived from `Shape` and overrides the `draw()` method to provide a specific implementation for drawing a circle.
3. In the `main` function, objects of both base (`Shape`) and derived (`Circle`) classes are created.
4. Base class pointers (`Shape*`) are used to point to objects of both classes, demonstrating polymorphism.
5. When calling the `draw()` method through the base class pointers, the appropriate overridden method is executed based on the actual type of the object.

Key Points:

- Method overriding requires the use of the `virtual` keyword in the base class method declaration.
- The `override` keyword in the derived class is optional but helps catch errors if the method is not actually overriding a base class method.
- Overriding allows a derived class to extend or modify the behavior of the base class method without changing its interface.

Method overriding is a powerful mechanism that enhances code flexibility and supports the

Reflection MCQs:

Q1. What is inheritance in C++?

A) A process of creating a new class from an existing class
B) A method to increase program speed
C) A technique to secure data
D) A way to create templates
Correct Answer: A

Q2. Which access specifier allows members of a derived class to access members of a base class?

A) Private
B) Protected
C) Public
D) Friend
Correct Answer: C

Q3. What type of inheritance is applied when a class is derived from two or more base classes?

A) Single Inheritance
B) Multiple Inheritance
C) Multilevel Inheritance
D) Hierarchical Inheritance
Correct Answer: B

Q4. In C++, which keyword is used to inherit a class?

A) extends
B) implements
C) derives
D) : (colon)
Correct Answer: D

Q5. Which of the following is not a type of constructor that C++ automatically provides?

A) Copy constructor
B) Default constructor
C) Virtual constructor
D) Destructor
Correct Answer: C

Q6. If a member of a base class is declared protected, who can access it?

A) Only the base class
B) Both the base class and derived class
C) Any class that requests access
D) None of the above
Correct Answer: B

Q7. What is the outcome when a derived class has the same function as declared in its base class?

A) Compile-time error
B) The function in the base class is overridden
C) The function in the derived class is ignored
D) Runtime error
Correct Answer: B

Q8. What does multi-level inheritance refer to?

A) A class is derived from multiple base classes
B) A class is derived from another derived class
C) Multiple classes are derived from the same base class
D) All of the above
Correct Answer: B

Q9. In hierarchical inheritance:

A) A class is derived from multiple base classes
B) Multiple classes are derived from a single base class
C) A class is derived from another derived class
D) None of the above
Correct Answer: B

Q10. Which of the following statements is true about constructors in derived classes?

A) Constructors of the base class are not called automatically
B) Constructors of the base class are called automatically
C) Constructors of the base class can only be called explicitly
D) Derived classes do not have constructors
Correct Answer: B

Q11. What happens when a derived class is defined with a "private" inheritance from a base class?

A) Public and protected members of the base class become private members of the derived class
B) Public members of the base class become protected members of the derived class
C) The derived class cannot access any members of the base class
D) The derived class can access only the public members of the base class
Correct Answer: A

Q12. Can a derived class override private members of its base class?

A) Yes, but only if accessed through a public member function of the base class
B) No, private members cannot be accessed or overridden
C) Yes, by declaring them again in the derived class
D) No, because private members are not inherited
Correct Answer: B

Q13. Which of the following correctly represents the destructor's functionality in inheritance?

A) Only the base class destructor is called
B) Only the derived class destructor is called
C) Both base and derived class destructors are called, starting with the derived class
D) Both base and derived class destructors are called, starting with the base class
Correct Answer: C

Q14. How is a protected member of a base class accessed in a derived class?

A) Directly, by name
B) Through a public member function of the base class
C) By making it public in the derived class
D) It cannot be accessed
Correct Answer: A

Q15. Which of the following is not possible in C++ inheritance?

A) A derived class with multiple base classes
B) A base class being derived from a derived class
C) A class being derived from itself

Code Exercise: Educational System Class Hierarchy

This is a code exercise that encompasses the concepts of inheritance, including public, protected, and private inheritance, as well as multiple and hierarchical inheritance. The exercise will guide you through creating a simple class hierarchy that models an educational system. This system will include a base class `Person`, from which `Student` and `Teacher` classes are derived, demonstrating single inheritance. Additionally, we will include a `TeachingAssistant` class that inherits from both `Student` and `Teacher`, showcasing multiple inheritance.

Objective: Implement a class hierarchy for an educational system consisting of persons, students, teachers, and teaching assistants.

Step 1: Define the `Person` class

- Create a base class named `Person` with the following members:
  - Protected data members: `name` (string) and `age` (int).
  - A parameterized constructor to initialize these members.
  - Public member functions to set and get the values of `name` and `age`.

Step 2: Implement the `Student` class

- Derive a class named `Student` from the `Person` class.
  - Add a private data member `studentID` (string).
  - Include a parameterized constructor that initializes the `Person` and `Student` attributes.
  - Implement public member functions to set and get the value of `studentID`.

Step 3: Implement the `Teacher` class

- Derive a class named `Teacher` from the `Person` class.
  - Add a private data member `teacherID` (string).
  - Include a parameterized constructor that initializes both `Person` and `Teacher` attributes.
  - Implement public member functions to set and get the value of `teacherID`.

Step 4: Implement the `TeachingAssistant` class

- Derive a class named `TeachingAssistant` from both `Student` and `Teacher` classes, demonstrating multiple inheritance.
  - Use an initializer list in the constructor to initialize the base classes.
  - Add a public function to display the teaching assistant's details.

Step 5: Main function

- In your `main()` function:
  - Instantiate an object of each class.
  - Set the attributes using the set functions or constructors.
  - Display the information of each object to validate the implementation.

Here is the starter code:

```cpp
#include <iostream>
#include <string>

// Step 1: Define the Person class
class Person {
protected:
    std::string name;
    int age;

public:
    // TODO: Implement a parameterized constructor for Person
    // TODO: Implement setters and getters for name and age
};



// Step 2: Implement the Student class
class Student : public Person {
private:
    std::string studentID;

public:
    // TODO: Implement a parameterized constructor for Student
    // TODO: Implement setters and getters for studentID
};
```

```
// Step 3: Implement the Teacher class
class Teacher : public Person {
private:
    std::string teacherID;

public:
    // TODO: Implement a parameterized constructor for Teacher
    // TODO: Implement setters and getters for teacherID
};



// Step 4: Implement the TeachingAssistant class
class TeachingAssistant : public Student, public Teacher {
public:
    // TODO: Implement a parameterized constructor for TeachingAssista
nt
    // TODO: Implement a function to display TeachingAssistant details
};



// Step 5: Main function
int main() {
    // TODO: Instantiate and initialize objects of Student, Teacher, a
nd TeachingAssistant
    // TODO: Display the information of each object to validate the im
plementation

    return 0;
}
```

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required program**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind

In [ ]: