CodInClub
Powered By BridgeLabz

Week7 /S2/ Lecture #: DSOOPS-31

Topics Covered

- Access Specifiers: Public, Private, and Protected (With Inheritance)
- Abstract Classes

Access Specifiers in C++ with Inheritance

In C++, access specifiers determine the accessibility of class members (variables and methods). The three main specifiers are:

- public: Members are accessible from anywhere.
- protected: Members are accessible within the class and its derived classes.
- private: Members are accessible only within the class itself.

Member Accessibility Without Inheritance

Specifier	**Access Inside Class**	**Access in Derived Class**	**Access Outside Class**
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	Yes	No	No

Inheritance Access Specifiers

When a class is inherited, the inheritance access specifier (`public`, `protected`, or `private`) affects how the inherited members' access is transformed in the derived class.

Let `Base` be the parent class and `Derived` the child class:

Base Class Members	**Public Inheritance**	**Protected Inheritance**	**Private Inheritance**
public	public	protected	private
protected	protected	protected	private
private	inaccessible	inaccessible	inaccessible

Note: Private members of the base class are never inherited directly; they are inaccessible in the derived class regardless of inheritance type.

Examples

```
class Base {
public:
    int pub = 1;
protected:
    int prot = 2;
private:
    int pvt = 3;
};

class PublicDerived : public Base {
public:
```

```
    void show() {
        std::cout << "public: " << pub << std::endl;     // accessible
        std::cout << "protected: " << prot << std::endl;  // accessible
        // std::cout << pvt; // error: not accessible
    }
};

class ProtectedDerived : protected Base {
public:
    void access() {
        std::cout << "public: " << pub << std::endl;     // accessible but as protected
        std::cout << "protected: " << prot << std::endl;  // accessible
    }
};

class PrivateDerived : private Base {
public:
    void access() {
        std::cout << "public: " << pub << std::endl;     // accessible but as private
        std::cout << "protected: " << prot << std::endl;  // accessible
    }
};

int main() {
    PublicDerived pubObj;
    std::cout << pubObj.pub << std::endl;  // OK, public is public
```

```
    ProtectedDerived protObj;
    // std::cout << protObj.pub << std::endl; // Error:
inaccessible (protected in Derived)

    PrivateDerived privObj;
    // std::cout << privObj.pub << std::endl; // Error:
inaccessible (private in Derived)

    return 0;
}
```

Summary Table for Inheritance Access Effects

Inheritance Type	**Base's Public Members Become**	**Base's Protected Members Become**	**Access to Base's Private Members**
Public	Public	Protected	No access
Protected	Protected	Protected	No access
Private	Private	Private	No access

Abstract Classes in C++

An abstract class is a class that cannot be instantiated on its own and is designed to be a base class. It usually contains at least one pure virtual function, forcing derived classes to provide implementations.

Pure Virtual Function

- Declared by assigning `= 0` to the function declaration.
- Example: `virtual void display() = 0;`

Purpose

- To define an interface for derived classes.
- To enforce that derived classes implement specific behaviors.

Syntax Example

```
class Shape {
public:
    // Pure virtual function makes this class abstract
    virtual void draw() = 0;

    void move() {
        std::cout << "Moving shape" << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing Circle" << std::endl;
    }
};

int main() {
    // Shape s; // Error: cannot instantiate abstract class
```

```
    Circle c;
    c.draw();  // Output: Drawing Circle
    c.move();  // Output: Moving shape
    return 0;
}
```

Key Points About Abstract Classes

- Cannot create objects of an abstract class.
- Any class with one or more pure virtual functions becomes abstract.
- Derived classes must override all pure virtual functions to be instantiable.
- Abstract classes can contain regular member functions with implementations.
- Used widely to implement runtime polymorphism via pointers or references to base class.

Practice Problems and Activities

Easy 1

Implement a class with all three access specifiers.

- Write a class `Example` with a public, protected, and private member variable.
- Write a member function inside the class that prints all three values.
- In `main()`, create an object and attempt to access each member directly. Print which are accessible.

Easy 2

Demonstrate public inheritance and member accessibility.

- Create a base class `Base` with one public and one protected member.
- Create a derived class `Derived` using public inheritance.

- Write a member function in `Derived` that prints both members.
- In `main()`, instantiate `Derived` and call the member function, then try to access both members directly.

Medium

Show protected inheritance and the impact on member accessibility.

- Write a base class `Alpha` with public and protected members.
- Derive a class `Beta` using protected inheritance.
- Write a member function in `Beta` that accesses the inherited members.
- In `main()`, create an object of `Beta` and try to access both members directly.

Hard

Abstract class and runtime polymorphism.

- Create an abstract class `Animal` with a pure virtual function `makeSound()`.
- Derive classes `Dog` and `Cat` and implement `makeSound()` for both.
- In `main()`, declare a pointer of type `Animal*`; assign it to objects of `Dog` and `Cat` and call `makeSound()` via the pointer to demonstrate polymorphism.
- Attempt to instantiate `Animal` and show/comment on the error.

Wrap-Up & Key Takeaways

- Access specifiers impact both direct and inherited member accessibility.
- Abstract classes and pure virtual functions help design interfaces and enforce behavior through inheritance.
- Practice with these patterns improves understanding of C++ class design.