جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الامن السيبراني

DEPARTMENT OF CYBER SECURITY

**SUBJECT:**

**OBJECT ORIENTED PROGRAMMING (OOP)**

**CLASS:**

**SECOND**

**LECTURER:**

**DR. ABDULKADHEM A. ABDULKADHEM**

# LECTURE (9):

## Inheritance in OOP (part 1)

# 1. Introduction to Inheritance

- **Definition**:
  Inheritance is a mechanism in OOP that allows one class (derived class) to inherit the properties and behaviors (members and functions) of another class (base class). The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process. The inheritance relationship is shown in Figure 1.

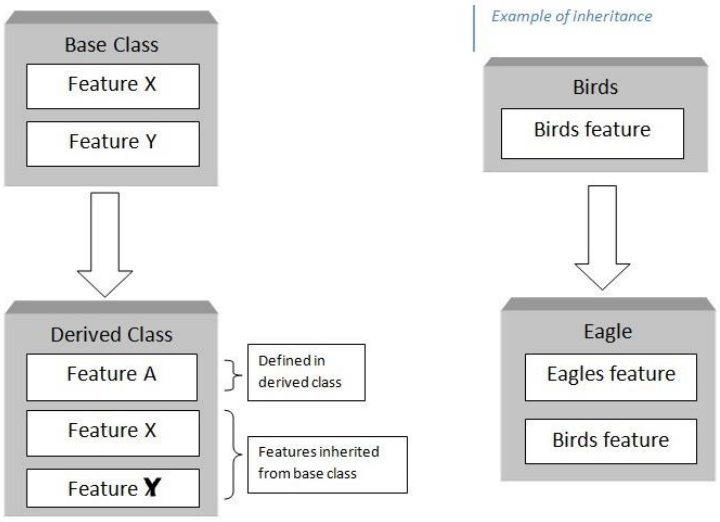



**Figure 1:** ***Inheritance.***

- **Why Use Inheritance?**
  - Promotes **code reusability**.
  - Establishes a **hierarchical relationship** between classes.
  - Enables **polymorphism**.

An important result of reusability is the ease of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

# 2. Derived Class and Base Class

***There are two main reasons that we might not want to modifying the base class.***

1) First, the base class works very well and has undergone many hours of testing and debugging.
2) Second reason for not modifying the base class: We might not have access to its source code, especially if it was distributed as part of a class library.

- **Syntax**:

```
class DerivedClass : accessSpecifier BaseClass {
  // Additional members and functions for the derived class
};
```

- **Key Points**:
  - The **accessSpecifier** can be `public`, `protected`, or `private`.
  - The derived class can access **public** and **protected** members of the base class, depending on the access specifier used.
- **Example**:

```
#include <iostream>
using namespace std;

// Base Class
class Base {
protected:
    int baseValue;
public:
    Base(int value) : baseValue(value) {}
    void showBaseValue() {
        cout << "Base value: " << baseValue << endl;
    }
};
// Derived Class
class Derived : public Base {
public:
    Derived(int value) : Base(value) {}
    void doubleBaseValue() {
        baseValue *= 2;
        cout << "Doubled Base value: " << baseValue << endl;
    }
};
int main() {
    Derived d(10);
    d.showBaseValue();
    d.doubleBaseValue();
    return 0;
}
```

```
Output:
Base value: 10
Doubled Base value: 20
```

## 3. The `protected` Access Specifier

We have increased the functionality of a class without modifying it. Let's first review what we know about the access specifies private and public. A member function of a class can always access class members, whether they are public or private. But an object declared externally can only invoke public members of the class. Private members are, well, private. This is shown in Figure 2.

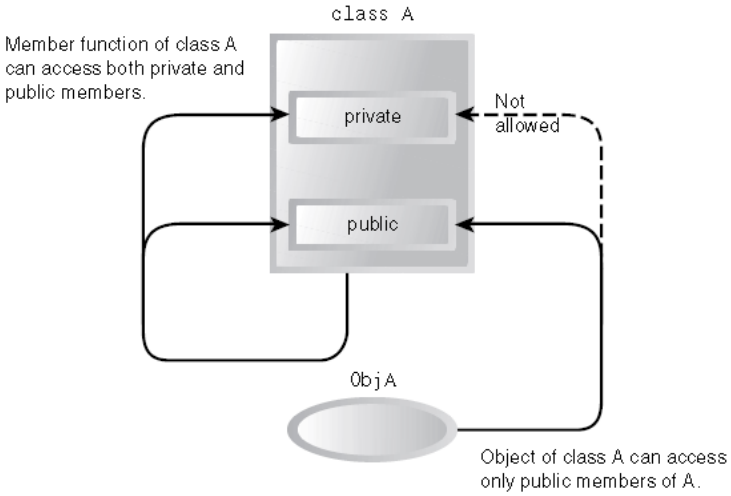



**Figure 2: Access *specifiers without inheritance***

A protected member, on the other hand, can be accessed by member functions in its own class and in any class derived from its own class. It can't be accessed from functions outside these classes, such as main(). The situation is shown in Figure 3.

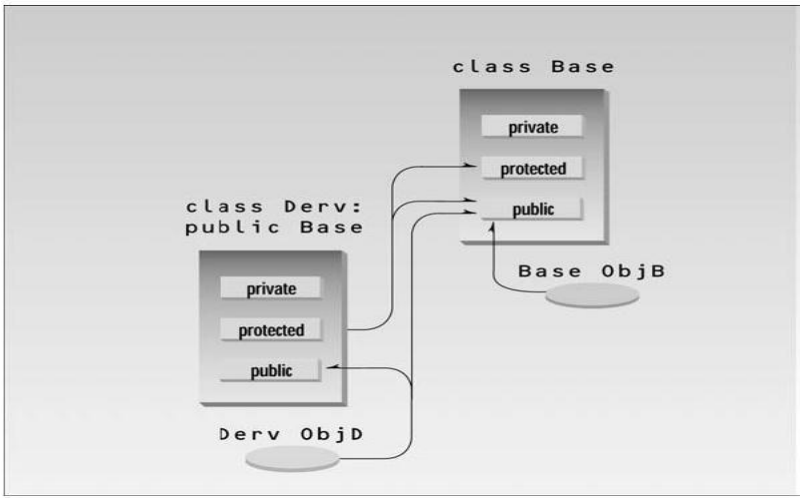
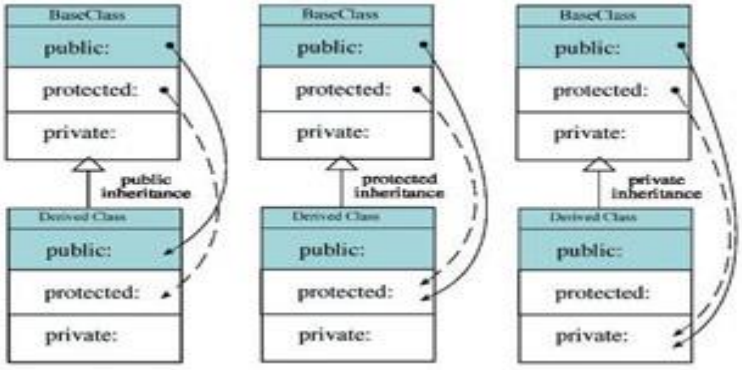



**Figure 3:Access specifiers with inheritance**

## 4. Inheritance and Accessibility

In **C++ inheritance**, the accessibility of members (variables and functions) of the base class depends on both the **access specifiers** of those members in the base class (`public`, `protected`, `private`) and the **type of inheritance** (`public`, `protected`, or `private`). Here's how it works:




**Case 1: Public Inheritance**

When a derived class inherits from a base class using `public` inheritance:

- **Public members** of the base class remain `public` in the derived class.
- **Protected members** of the base class remain `protected` in the derived class.
- **Private members** of the base class are **not directly accessible** in the derived class.

**Key Point:**

Private members of the base class are **not inherited in a way that allows direct access**. However, they are still part of the base class and can be accessed **indirectly** through public or protected member functions provided by the base class.

**Case 2: Protected Inheritance**

When a derived class inherits using `protected` inheritance:

- **Public members** of the base class become `protected` in the derived class.
- **Protected members** of the base class remain `protected` in the derived class.
- **Private members** of the base class remain inaccessible directly in the derived class.

**Key Point:**

This inheritance type is useful when you want to limit access to the inherited public members but still allow derived classes to use them.

**Case 3: Private Inheritance**

When a derived class inherits using `private` inheritance:

- Both **public** and **protected** members of the base class become `private` in the derived class.
- **Private members** of the base class remain inaccessible directly in the derived class.

**Key Point:**

This inheritance type ensures that the base class's interface is completely hidden from outside classes, and access is limited to the derived class's internal implementation.

**Can a Derived Class Access Private Members of the Base Class?**

No, **private members** of a base class are not directly accessible by a derived class, regardless of the inheritance type. Private members are strictly encapsulated within the base class. However, the derived class can access private members **indirectly** through public or protected member functions provided by the base class.

**Example**: Here's a complete example that demonstrates **all types of inheritance** (`public`, `protected`, and `private`) and how access specifiers (`public`, `protected`, and `private`) behave under each type:

```
#include <iostream>
using namespace std;
class Base {
private:
    int privateVar;  // Private: not accessible directly in any derived class
protected:
    int protectedVar; // Protected: accessible in derived classes
public:
    int publicVar;    // Public: accessible to everyone
    Base() : privateVar(10), protectedVar(20), publicVar(30) {}
                      // Public function to access privateVar
    int getPrivateVar() const {
        return privateVar;
    }
    void setPrivateVar(int value) {
        privateVar = value;
    }
};
// Derived class with Public Inheritance
class PublicDerived : public Base {
public:
    void display() {
        // cout << privateVar; // Error: privateVar is inaccessible
        cout << "Public Inheritance:" << endl;
        cout << "ProtectedVar: " << protectedVar << endl; // Accessible
        cout << "PublicVar: " << publicVar << endl;       // Accessible
        cout << "PrivateVar (via getter): " << getPrivateVar() << endl;
    }
};
// Derived class with Protected Inheritance
class ProtectedDerived : protected Base {
public:
    void display() {
        cout << "Protected Inheritance:" << endl;
        // cout << privateVar; // Error: privateVar is inaccessible
        cout << "ProtectedVar: " << protectedVar << endl; // Accessible
```

```
        cout << "PublicVar: " << publicVar << endl;   // Accessible (now protected)
    }
};
// Derived class with Private Inheritance
class PrivateDerived : private Base {
public:
    void display() {
        cout << "Private Inheritance:" << endl;
        // cout << privateVar; // Error: privateVar is inaccessible
        cout << "ProtectedVar: " << protectedVar << endl; // Accessible
        cout << "PublicVar: " << publicVar << endl;    // Accessible (now private)
    }
};
// Trying to access members from outside the derived classes
void accessTest() {
    PublicDerived publicObj;
    cout << "\nAccess Test - Public Inheritance:" << endl;
    publicObj.display();
    cout << "Access publicVar: " << publicObj.publicVar << endl; //
Accessible
    // publicObj.protectedVar; // Error: protectedVar is inaccessible
    // publicObj.privateVar;   // Error: privateVar is inaccessible
    ProtectedDerived protectedObj;
    cout << "\nAccess Test - Protected Inheritance:" << endl;
    protectedObj.display();
    // protectedObj.publicVar; // Error: publicVar is now protected
    // protectedObj.protectedVar; // Error: protectedVar is
inaccessible
    // protectedObj.privateVar;   // Error: privateVar is inaccessible

    PrivateDerived privateObj;
    cout << "\nAccess Test - Private Inheritance:" << endl;
    privateObj.display();
    // privateObj.publicVar; // Error: publicVar is now private
    // privateObj.protectedVar; // Error: protectedVar is inaccessible
    // privateObj.privateVar;   // Error: privateVar is inaccessible
}
int main() {
    accessTest();
    return 0;
}
```

**Output:**

```
Access Test - Public Inheritance:
Public Inheritance:
ProtectedVar: 20
PublicVar: 30
PrivateVar (via getter): 10
Access publicVar: 30

Access Test - Protected Inheritance:
Protected Inheritance:
ProtectedVar: 20
PublicVar: 30

Access Test - Private Inheritance:
Private Inheritance:
ProtectedVar: 20
PublicVar: 30
```

**5. Dangers of protected**

You should know that there's a disadvantage to making class members protected. Say you've written a class library, which you're distributing to the public. Any programmer who buys this library can access protected members of your classes simply by deriving other classes from them. This makes protected members considerably less secure than private members. To avoid corrupted data, it's often safer to force derived classes to access data in the base class using only public functions.

## 6. Overriding Member Functions

Overriding allows a **derived class** to provide a new implementation for a function that already exists in the **base class**. The overridden function in the derived class must have the same **name**, **return type**, and **parameters** as the function in the base class.

- **Key Points**:
  - The base class function should be declared as `virtual` to enable overriding.
  - Overriding is a cornerstone of **runtime polymorphism**.
  -
- **Example**:

```
#include <iostream>
using namespace std;

class Base {
public:
    virtual void display() {      // Virtual function in the base class
        cout << "Base class display function" << endl;
    }
};
class Derived : public Base {
public:
    void display() override {     // Overriding the base class function
        cout << "Derived class display function" << endl;
    }
};
int main() {
    Base* basePtr;
    Derived derivedObj;
    basePtr = &derivedObj;

    basePtr->display();    // Calls the derived class's display function
    return 0;
}
Output:
Derived class display function
```

## 7. Accessing Base Class Members

- **Using the Scope Resolution Operator (`::`)**: A derived class can explicitly call functions or access members of the base class using the scope resolution operator.

- **Example**:

```
class Base {
public:
    void show() {
        cout << "Base class show function" << endl;
    }
};
class Derived : public Base {
public:
    void show() {
        cout << "Derived class show function" << endl;
        Base::show();                    // Calling the base class show function
    }
};
int main() {
    Derived obj;
    obj.show();
    return 0;
}
```

```
Output:
Derived class show function
Base class show function
```

## 8. Types of Inheritance

Briefly mention the types of inheritance (next lecture):

1. **Single Inheritance**: One base class, one derived class.
2. **Multiple Inheritance**: Derived class inherits from multiple base classes.
3. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.
4. **Multilevel Inheritance**: A class inherits from a derived class.
5. **Hybrid Inheritance**: A combination of two or more types of inheritance.