

C++ Inheritance: Comprehensive Notes

Table of Contents

| | |
|--|--|
| 1. Introduction to Inheritance | |
| 2. Basic Inheritance Syntax | |
| 3. Access Specifiers in Inheritance | |
| 4. Types of Inheritance | |
| • Single Inheritance | |
| • Multiple Inheritance | |
| • Multilevel Inheritance | |
| • Hierarchical Inheritance | |
| • Hybrid Inheritance | |
| 5. Function Overriding | |
| 6. Constructor and Destructor in Inheritance | |
| 7. Virtual Functions | <div>Notes will be provided in :provided in : C++ Inheritance Notes Part II</div> |
| 8. Abstract Classes and Pure Virtual Functions | |
| 9. Virtual Inheritance and Diamond Problem | |
| 10. Object Slicing | |
| 11. Protected Members | |
| 12. Final Specifier | |
| 13. Override Specifier | |
| 14. Best Practices for C++ Inheritance | |
| 15. Real-world Example: Shape Hierarchy | |

Introduction to Inheritance

Inheritance is a core concept in object-oriented programming that allows a class (derived class) to inherit properties and behaviors from another class (base class). This promotes code reuse and establishes an "is-a" relationship between classes.

Key Benefits of Inheritance

- **Code Reusability:** Reuse attributes and methods from existing classes
- **Extensibility:** Add new features without modifying existing code
- **Hierarchical Classification:** Organize classes in a logical hierarchy
- **Polymorphism:** Enable runtime behavior through method overriding

Basic Inheritance Syntax

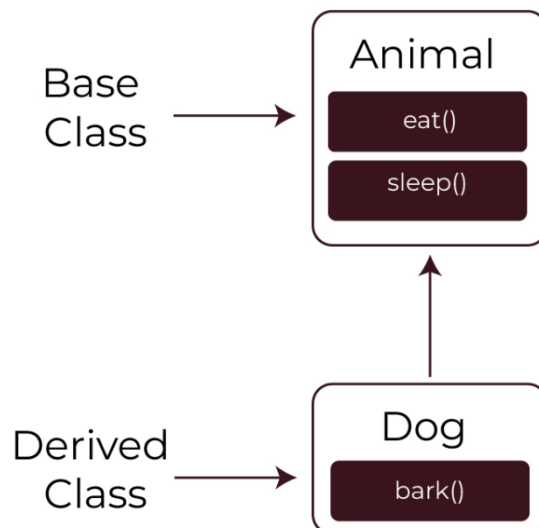
```
class BaseClass {  
    // Base class members  
};
```

```
class DerivedClass : access-specifier BaseClass {  
    // Derived class members  
};
```

Where `access-specifier` can be:

- `public`: Public and protected members of the base class remain public and protected in the derived class
- `protected`: Public and protected members of the base class become protected in the derived class
- `private`: Public and protected members of the base class become private in the derived class

Simple Example



```

#include <iostream>
using namespace std;
// Base class
class Animal {
protected:
    string name;
    int age;
public:
    Animal(string name, int age) : name(name), age(age) {}

    void eat() {
        cout << name << " is eating." << endl;
    }

    void sleep() {
        cout << name << " is sleeping." << endl;
    }
};
// Derived class
class Dog : public Animal {
private:
    string breed;
public:
    Dog(string name, int age, string breed)
        : Animal(name, age), breed(breed) {}

    void bark() {
        cout << name << " (a " << breed << ") is barking." << endl;
    }

    void display() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Breed: " << breed << endl;
    }
};

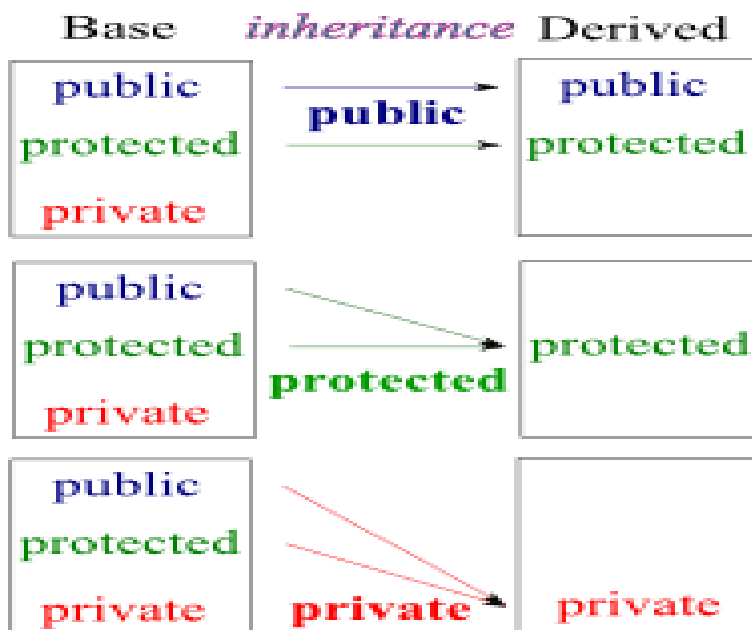
int main() {
    Dog dog("Max", 3, "German Shepherd");
    dog.display();
    dog.eat();           // Inherited from Animal
    dog.sleep();         // Inherited from Animal
    dog.bark();          // Defined in Dog
    return 0;
}

```

Access Specifiers in Inheritance –**Very Important

The access specifier used in inheritance affects how the base class members are accessible in the derived class:

| Base class member access specifier | Type of Inheritance | | |
|------------------------------------|-------------------------|-------------------------|-------------------------|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not accessible (Hidden) | Not accessible (Hidden) | Not accessible (Hidden) |

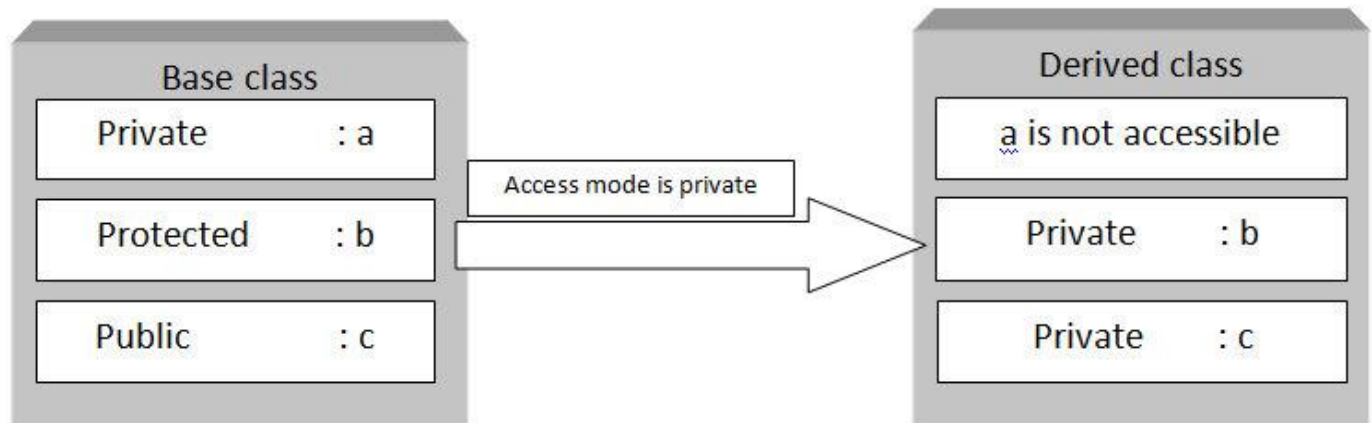


private access specifier

If private access specifier is used while creating a class, then the public and protected data members of the base class become the private member of the derived class and private member of base class remains private.

In this case, the members of the base class can be used only within the derived class and cannot be accessed through the object of derived class whereas they can be accessed by creating a function in the derived class.

Following block diagram explain how data members of base class are inherited when derived class access mode is private.

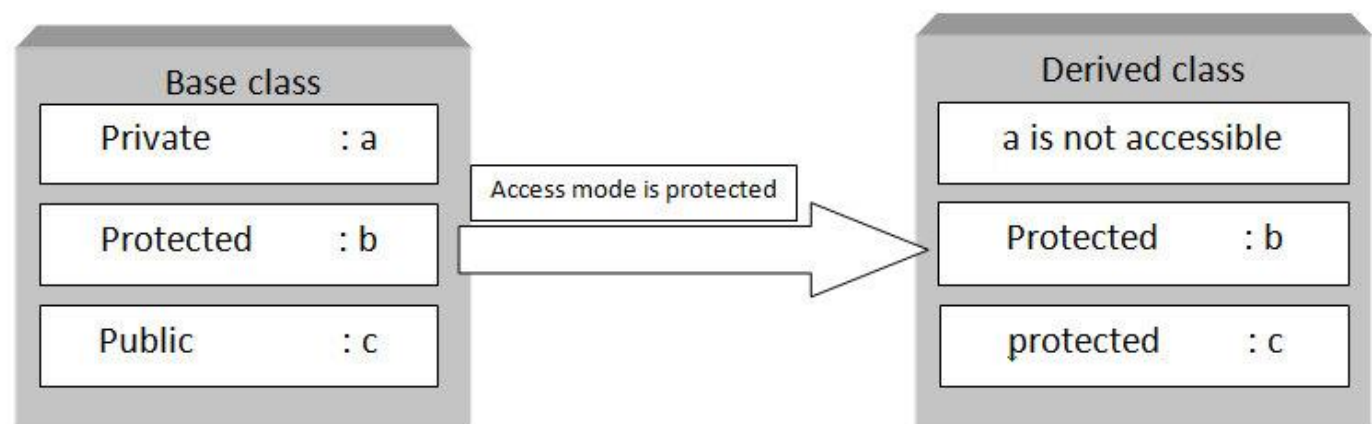


Protected Access Specifier

If protected access specifier is used while deriving class then the public and protected data members of the base class becomes the protected member of the derived class and private member of the base class are inaccessible.

In this case, the members of the base class can be used only within the derived class as protected members except for the private members.

Following block diagram explain how data members of base class are inherited when derived class access mode is protected.

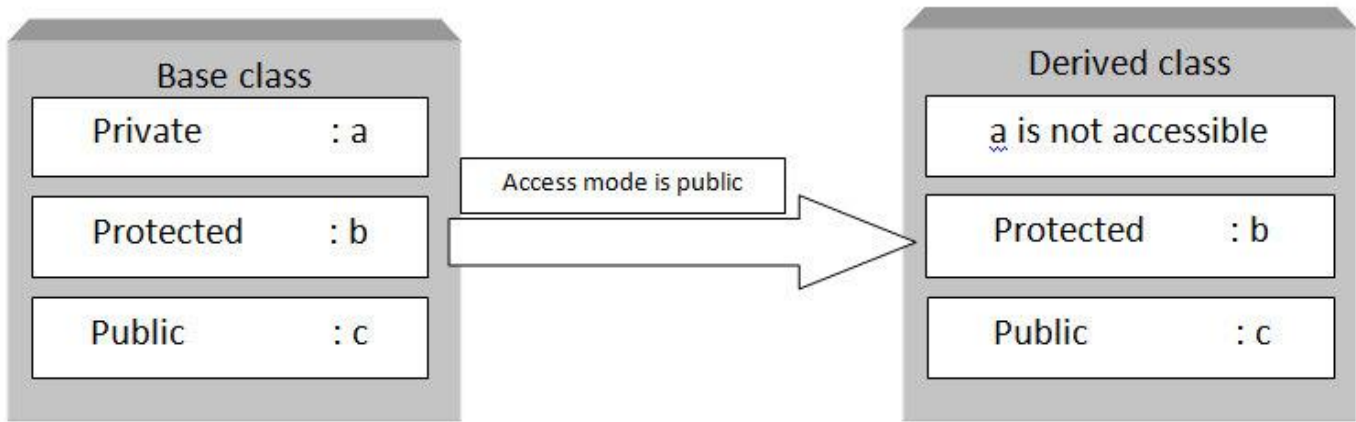


public access specifier

If public access specifier is used while deriving class then the public data members of the base class becomes the public member of the derived

class and protected members becomes the protected in the derived class but the private members of the base class are inaccessible.

Following block diagram explain how data members of base class are inherited when derived class access mode is public



EXTRA for reference:

| Base class member access specifier | Type of inheritance | | |
|------------------------------------|--|--|--|
| | public inheritance | protected inheritance | private inheritance |
| public | public in derived class Can be accessed directly by member, friend and nonmember functions | protected in derived class Can be accessed directly by member and friend functions | private in derived class Can be accessed directly by member and friend functions |
| protected | protected in derived class Can be accessed directly by member and friend functions | protected in derived class Can be accessed directly by member and friend functions | private in derived class Can be accessed directly by member and friend functions |
| private | Hidden in derived class Can be accessed by member and friend functions through public or protected member of the base class | Hidden in derived class Can be accessed by member and friend functions through public or protected member functions of the base class | Hidden in derived class Can be accessed by member and friend functions through public or protected member functions of the base class |

Example Demonstrating Access Specifiers

```
#include <iostream>
using namespace std;
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
public:
    Base() : publicVar(1), protectedVar(2), privateVar(3) {}

    void display() {
        cout << "Base class: " << endl;
        cout << "Public: " << publicVar << endl;
        cout << "Protected: " << protectedVar << endl;
        cout << "Private: " << privateVar << endl;
    }
};
class PublicDerived : public Base {
public:
    void access() {
        cout << "PublicDerived accessing:" << endl;
        cout << "Public: " << publicVar << endl;           // Accessible
        cout << "Protected: " << protectedVar << endl;       // Accessible
        // cout << "Private: " << privateVar << endl;         // Not accessible
    }
};
class ProtectedDerived : protected Base {
public:
    void access() {
        cout << "ProtectedDerived accessing:" << endl;
        cout << "Public: " << publicVar << endl;           // Accessible
        cout << "Protected: " << protectedVar << endl;       // Accessible
        // cout << "Private: " << privateVar << endl;         // Not accessible
    }
};
class PrivateDerived : private Base {
public:
    void access() {
        cout << "PrivateDerived accessing:" << endl;
        cout << "Public: " << publicVar << endl;           // Accessible
        cout << "Protected: " << protectedVar << endl;       // Accessible
        // cout << "Private: " << privateVar << endl;         // Not accessible
    }
};
```

```

    }
};

int main() {
    PublicDerived pubDer;
    pubDer.publicVar = 10;           // Accessible
    // pubDer.protectedVar = 20;    // Not accessible
    // pubDer.privateVar = 30;     // Not accessible
    pubDer.access();
    pubDer.display();                // Accessible

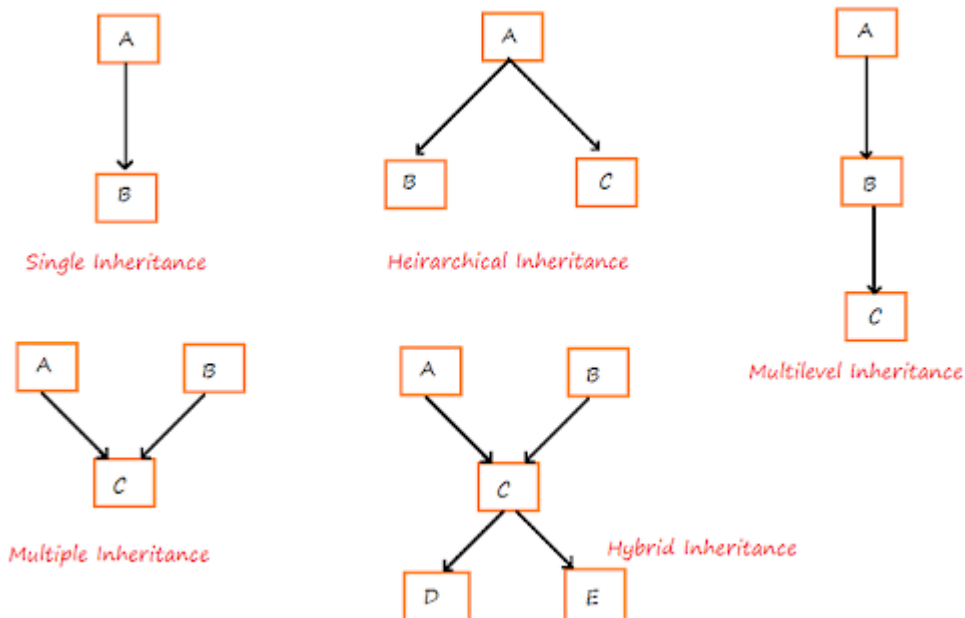
    ProtectedDerived protDer;
    // protDer.publicVar = 10;      // Not accessible due to protected inheritance
    // protDer.protectedVar = 20;   // Not accessible
    // protDer.privateVar = 30;    // Not accessible
    protDer.access();
    // protDer.display();           // Not accessible due to protected inheritance

    PrivateDerived privDer;
    // privDer.publicVar = 10;      // Not accessible due to private inheritance
    // privDer.protectedVar = 20;   // Not accessible
    // privDer.privateVar = 30;    // Not accessible
    privDer.access();
    // privDer.display();           // Not accessible due to private inheritance

    return 0;
}

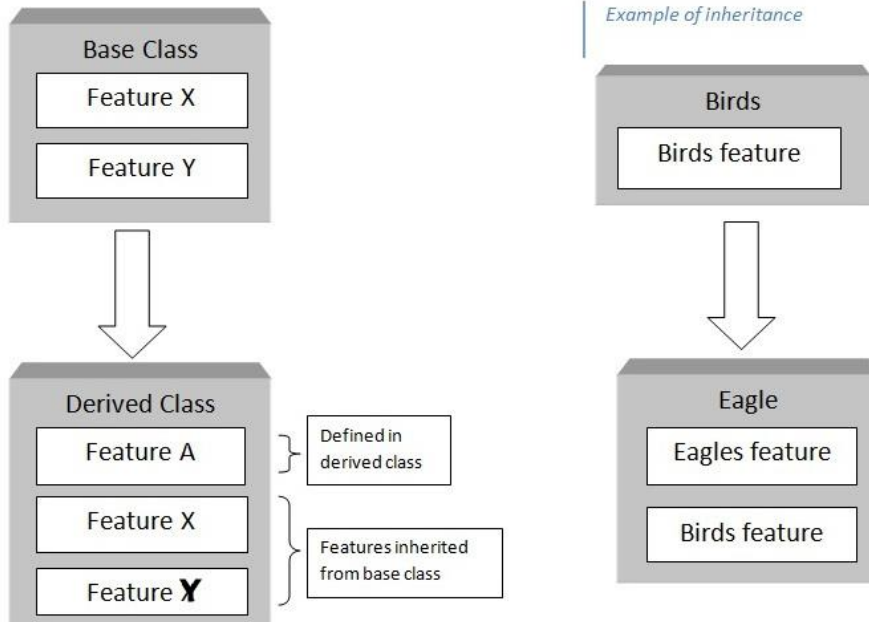
```

Types of Inheritance



Single Inheritance

A derived class inherits from only one base class.



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

class Parent {
public:
    void display() {
        cout << "This is the parent class." << endl;
    }
};

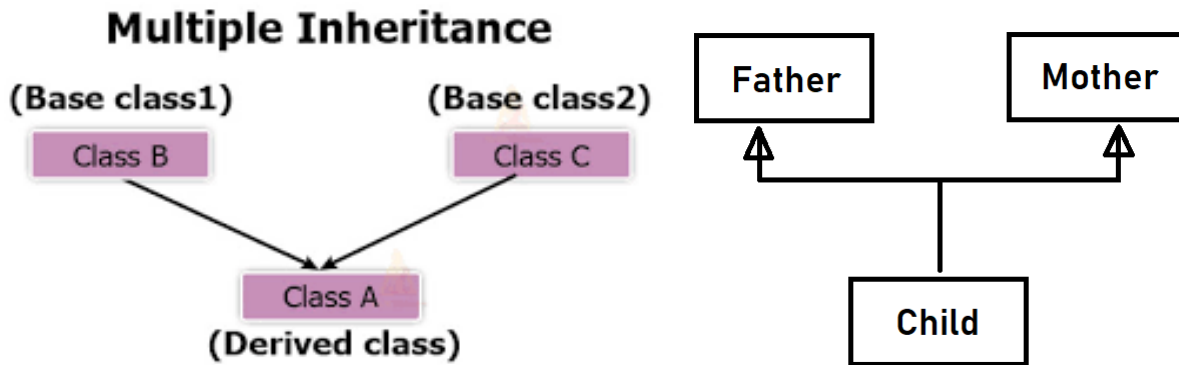
class Child : public Parent {
public:
    void show() {
        cout << "This is the child class." << endl;
    }
};

int main() {
    Child obj;
    obj.display(); // From Parent class
    obj.show();    // From Child class
    return 0;
}
```

Multiple Inheritance

A derived class inherits from multiple base classes.

Diagram:



```
#include <iostream>
using namespace std;
class Father {
public:
    void fishing() {
        cout << "Father enjoys fishing." << endl;
    }
};

class Mother {
public:
    void cooking() {
        cout << "Mother enjoys cooking." << endl;
    }
};

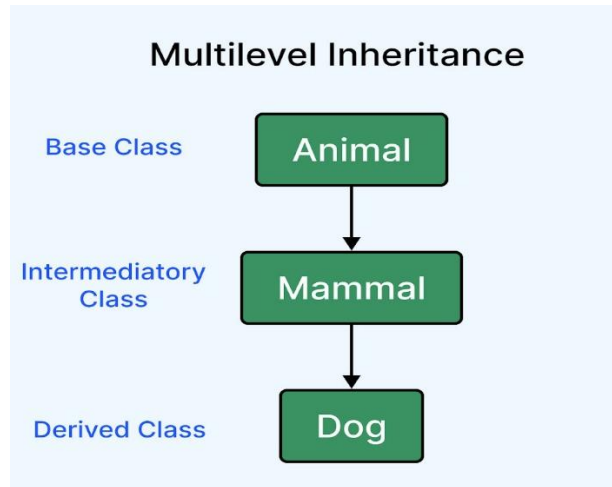
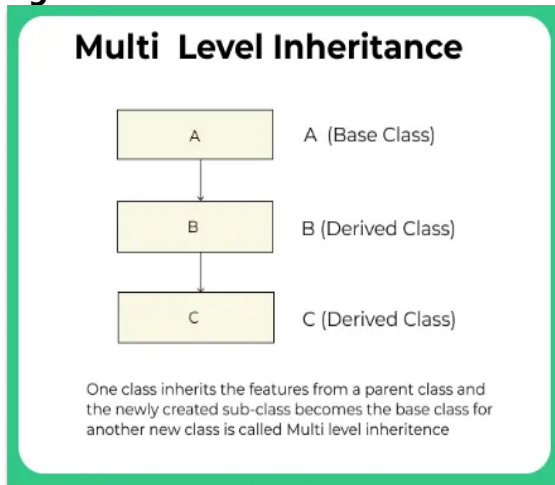
class Child : public Father, public Mother {
public:
    void playing() {
        cout << "Child enjoys playing." << endl;
    }
};

int main() {
    Child child;
    child.fishing(); // From Father class
    child.cooking(); // From Mother class
    child.playing(); // From Child class
    return 0;
}
```

Multilevel Inheritance

A derived class inherits from a class, which in turn inherits from another class.

Diagram:



```
#include <iostream>
using namespace std;
class Animal {
public:
    void eat() {
        cout << "Animal eats food." << endl;
    }
};

class Mammal : public Animal {
public:
    void breathe() {
        cout << "Mammal breathes air." << endl;
    }
};

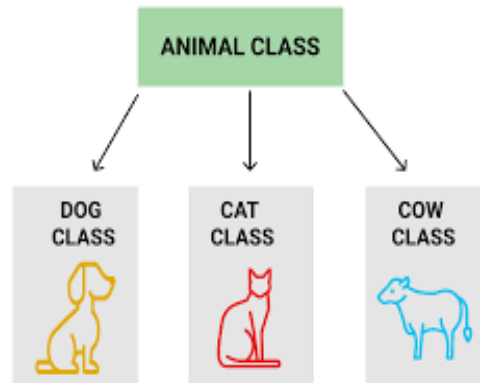
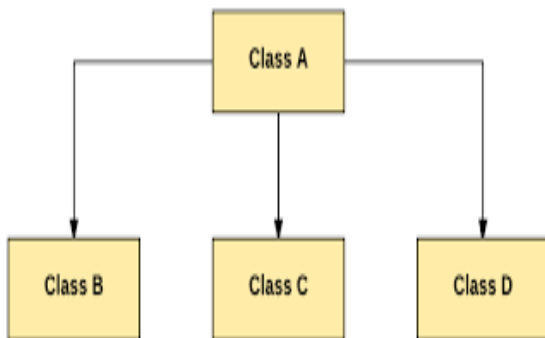
class Dog : public Mammal {
public:
    void bark() {
        cout << "Dog barks: Woof!" << endl;
    }
};

int main() {
    Dog dog;
    dog.eat();    // From Animal class
    dog.breathe(); // From Mammal class
    dog.bark();   // From Dog class
    return 0;
}
```

Hierarchical Inheritance

Multiple derived classes inherit from a single base class.

Diagram:



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

class Animal {
public:
    void eat() {
        cout << "Animal eats food." << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks: Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void meow() {
        cout << "Cat meows: Meow!" << endl;
    }
};

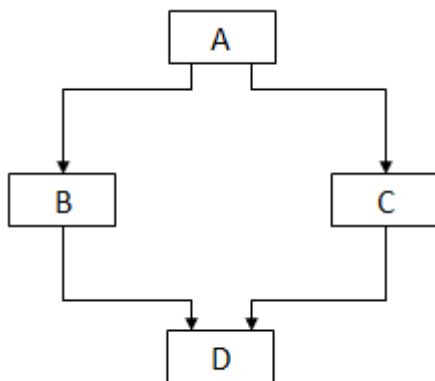
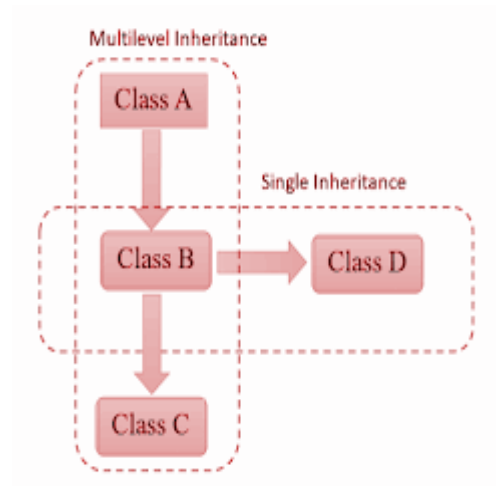
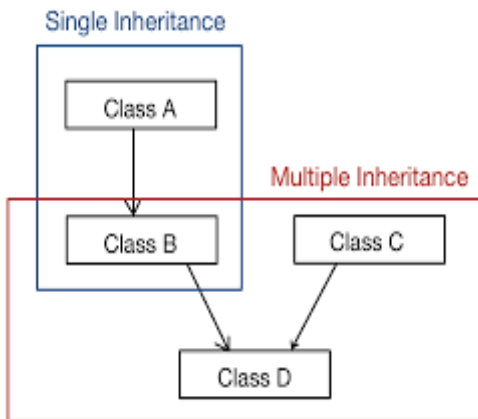
class Cow : public Animal {
public:
    void moo() {
        cout << "Cow moos: Moo!" << endl;
    }
};
```

```
int main() {  
    Dog dog;  
    Cat cat;  
    Cow cow;  
    dog.eat(); // From Animal class  
    dog.bark(); // From Dog class  
  
    cat.eat(); // From Animal class  
    cat.meow(); // From Cat class  
  
    cow.eat(); // From Animal class  
    cow.moo(); // From Cow class  
    return 0;  
}
```

Hybrid Inheritance

Combination of multiple types of inheritance.

Diagram:



Hybrid Inheritance

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

```
class A {
protected:
    int a;
public:
    A() : a(10) {}
};
```

```
class B : public A {
protected:
    int b;
public:
    B() : b(20) {}
```

```
};
```

```
class C : public A {  
protected:  
    int c;  
public:  
    C() : c(30) {}  
};
```

```
class D : public B, public C {  
private:  
    int d;  
public:  
    D() : d(40) {}
```

// Error due to ambiguity: we have two copies of A's members

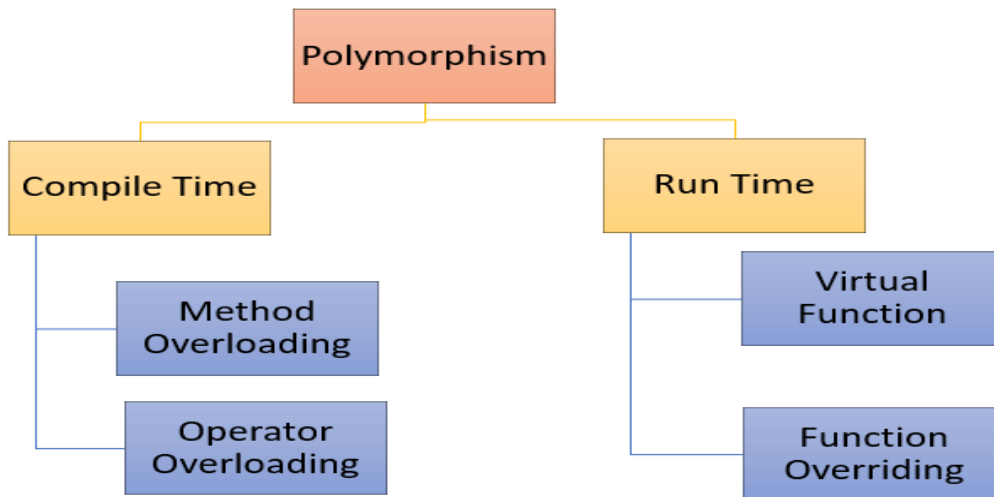
```
void display() {  
    // cout << "a = " << a << endl; // Ambiguous!  
    cout << "b = " << b << endl;  
    cout << "c = " << c << endl;  
    cout << "d = " << d << endl;  
}  
};
```

```
int main() {  
    D obj;  
    obj.display();  
    return 0;  
}
```

Function Overriding (Run time polymorphism)

Function overriding occurs when a derived class has a method with the same name and signature as a method in its base class. The derived class's method "overrides" the base class's method.

function overriding is a key mechanism for achieving **runtime polymorphism (also known as dynamic polymorphism)** in object-oriented programming languages like Java and C++.



Overriding vs. Overloading

| Comparison Criteria | Overriding | Overloading |
|------------------------|--|---|
| Method name | Must be the same. | Must be the same. |
| Argument list | Must be the same. | Must be different. |
| Return type | Can be the same type or a covariant type. | Can be different. |
| throws clause | Must not throw new checked exceptions. Can narrow exceptions thrown. | Can be different. |
| Accessibility | Can make it less restrictive, but not more restrictive. | Can be different. |
| Declaration context | A method can only be overridden in a subclass. | A method can be overloaded in the same class or in a subclass. |
| Method call resolution | The <i>runtime type</i> of the reference, i.e., the type of the object referenced at <i>runtime</i> , determines which method is selected for execution. | At compile time, the <i>declared type</i> of the reference is used to determine which method will be executed at runtime. |

| Aspect | Method Overloading | Method Overriding |
|-----------------------------|--|---|
| Definition | Having multiple methods in the same class with the same name, but differing in the number or type of parameters. | Occurs when a subclass provides a specific implementation for a method already defined in its superclass. |
| Real-Life Analogy | Imagine a phone with buttons. The same button can perform different functions based on how you press it. | Think of a car's accelerator pedal, where pressing it in different cars may produce varying speeds, but the action is the same. |
| Use Case | Providing multiple ways to perform a similar operation with different inputs. | Customizing the behavior behavior of a method inherited from a superclass to suit the needs of a subclass. |
| Method Signature | Differs in the number or type of parameters. | Must have the same method name, return type, and parameters (method signature). |
| Compile-Time Polymorphism | Resolved at compile-time. | Not applicable; it is related to runtime polymorphism. |
| Flexibility and Versatility | Increases code flexibility and versatility. | Specializes or customizes behavior behavior within an object-oriented hierarchy. |

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

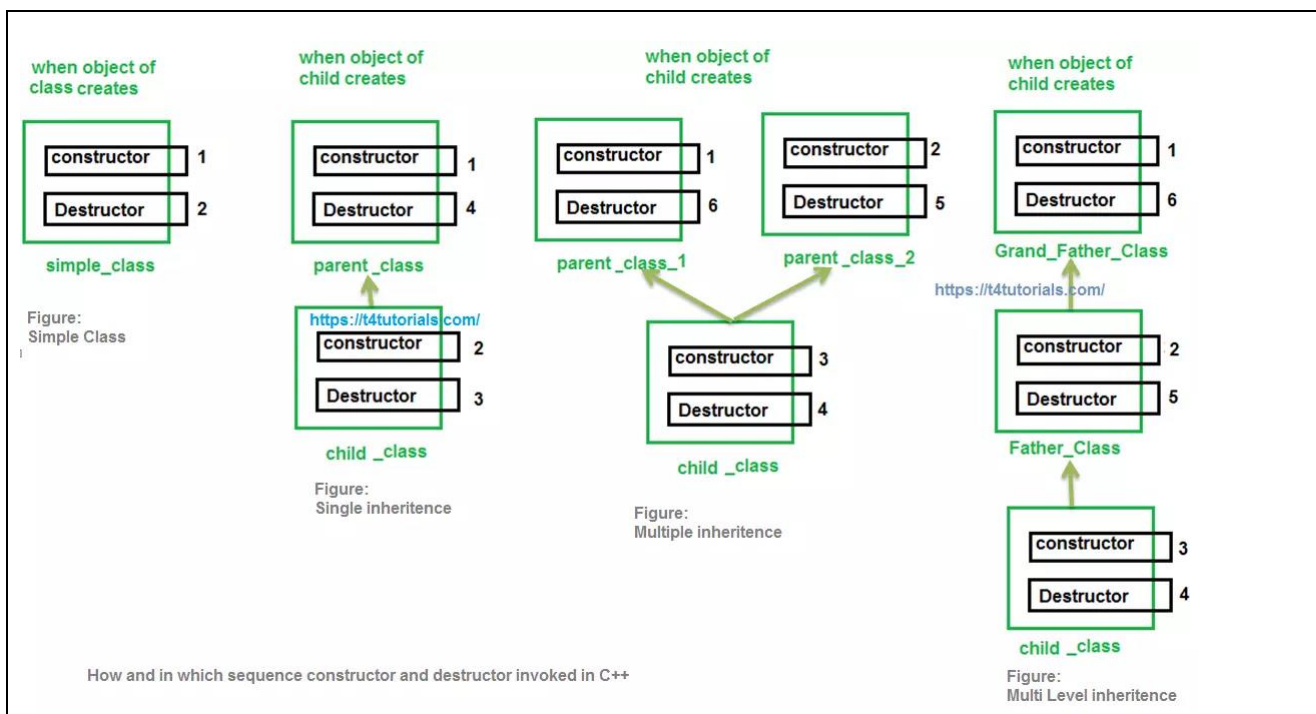
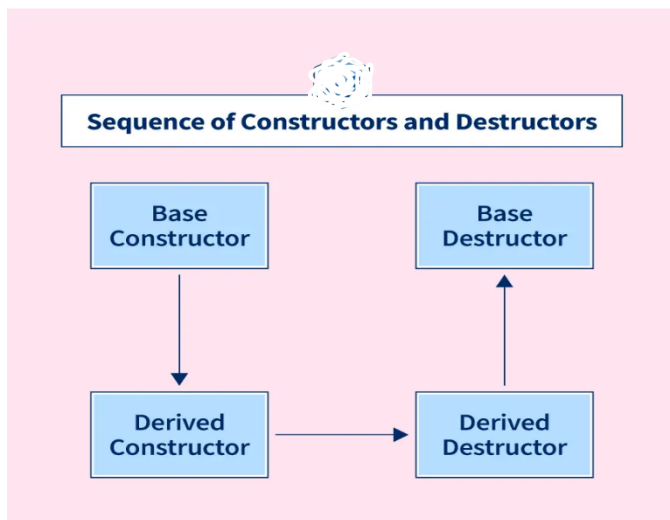
class Base {
public:
    void display() {
        cout << "Display method of Base class." << endl;
    }
};

class Derived : public Base {
public:
    // Override display() method
    void display() {
        cout << "Display method of Derived class." << endl;
    }
};
```

```
int main() {  
    Derived d;  
    d.display(); // Calls the Derived class's display method  
  
    // To call the Base class version:  
    d.Base::display(); // Explicitly calling Base class method  
  
    return 0;  
}
```

Constructor and Destructor in Inheritance

Constructors and destructors are called in a specific order in an inheritance hierarchy.



```
#include <iostream>
using namespace std;
class Base {
public:
    Base() {
        cout << "Base constructor called" << endl;
    }
    ~Base() {
        cout << "Base destructor called" << endl;
    }
};
```

```
class Derived : public Base {
public:
    Derived() {
        cout << "Derived constructor called" << endl;
    }

    ~Derived() {
        cout << "Derived destructor called" << endl;
    }
};

class DerivedMore : public Derived {
public:
    DerivedMore() {
        cout << "DerivedMore constructor called" << endl;
    }

    ~DerivedMore() {
        cout << "DerivedMore destructor called" << endl;
    }
};

int main() {
    DerivedMore obj;
    // Object creation is complete here
    // Object will be destroyed when it goes out of scope
    return 0;
}
```

Output:



```
Base constructor called
Derived constructor called
DerivedMore constructor called
DerivedMore destructor called
Derived destructor called
Base destructor called
```

Calling Base Class Constructor Explicitly

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

class Base {
private:
    int value;

public:
    Base(int v) : value(v) {
        cout << "Base constructor called with value: " << value << endl;
    }

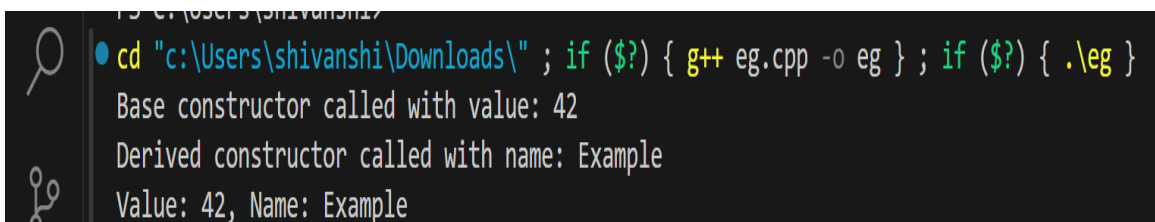
    int getValue() const {
        return value;
    }
};

class Derived : public Base {
private:
    string name;

public:
    // Call Base constructor explicitly with initialization list
    Derived(int v, string n) : Base(v), name(n) {
        cout << "Derived constructor called with name: " << name << endl;
    }

    void display() {
        cout << "Value: " << getValue() << ", Name: " << name << endl;
    }
};

int main() {
    Derived obj(42, "Example");
    obj.display();
    return 0;
}
```



```
PS C:\Users\shivanshi\Downloads> cd "c:\Users\shivanshi\Downloads\" ; if ($?) { g++ eg.cpp -o eg } ; if ($?) { .\eg }
Base constructor called with value: 42
Derived constructor called with name: Example
Value: 42, Name: Example
```