25|10|2023

Object Oriented Programming Class 02

1: Copy Constructor

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

class Student
{
private:
    string gf;

    void chatting(){
        cout<< "Chatting " << endl;
    }

public:
    int id;
    string name;
    int age;
    string gender;

    // Default CTOR: assign garbage value
    Student(){
        cout<< "Default ctor called" <<endl;
    }

    // Parameterized CTOR: assign sensible value
    Student(int _id, string _name, int _age, string _gender, string _gf){
        id = _id;
        name = _name;
        age = _age;
        gender = _gender;
        gf = _gf;
        cout<< "Parameterized ctor called for " << name <<endl;
    }

    void study(){
        cout<< "Studying" << endl;
    }
};
```

```cpp
int main(){

    Student s1(104,"Love",25,"Male","Lovely");

    Student s2;

    s2 = s1;

    cout<< s2.age <<" "<< s2.name << endl;

    return 0;
}
```

1. ***Source = s1,*** *Programmer added parameterized CTOR*
2. ***Destination : s2,*** *Compiler added default CTOR by default*
3. ***Destination = Source,*** *Now copied s1 into s2 (In this case, copy constructor added by compiler)*
4. ***Now s2 is accessing states*** *of s1 because s1 ke data ki copy s2 ke pass hai*

OUTPUT:

- ✓ Parameterized ctor called for Love
- ✓ Default ctor called
- ✓ 25 Love

{ Jab tak Hum Khudse Copy CTOR Nahi Banaye Tab Tak Bad practice mani jayegi }

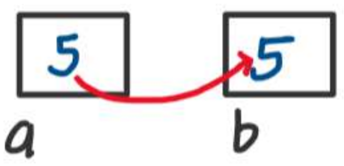
```cpp
int main(){

    Student s1(104,"Love",25,"Male","Lovely");

    Student s2 = s1;  // or s2(s1);

    cout<< s2.age <<" "<< s2.name << endl;

    return 0;
}
```

1 **Destination = Source,** *Now copied s1 into s2 (In this case, also copy constructor added by compiler)*

OUTPUT: *Parameterized ctor called for Love* *25 Love*

How to add copy CTOR by own?

jab main



a copied into b

```
int a = 5
int b = a;
```

Two integer copied

Dest. Src.

Jab main

S1 copied into S2

```
Student S1(Data);
Student S2 = S1;
```

Two student copied

Dest. Src.

COPY CTOR NAME

srcObj points to s2

REFERENCE TO AN OBJECT OF CLASS s1

this points to s2

```cpp
// Own copy CTOR: the source 's1' is coping into the destination 's2'
Student(const Student &srcObj){
    // Copy attributes from source object to the current object
    this->id = srcObj.id;
    this->name = srcObj.name;
    this->age = srcObj.age;
    this->gender = srcObj.gender;
    this->gf = srcObj.gf;
    cout<<"Copy CTOR called"<<endl;
}
```

COPY CTOR BODY {..}

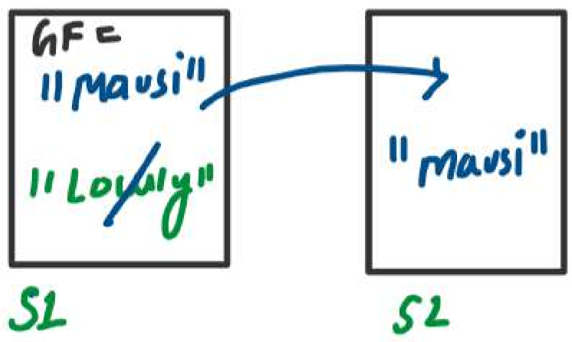
Why we need of copy constructor?

We should understand the concept of shallow and deep copy to understand of this question.

Why we need of const in copy constructor?

```cpp
// Own copy CTOR: the source 's1' is coping into the destination 's2'
Student(Student &srcObj){

    // Hacker
    srcObj.gf = "Mausi😝";

    // Copy attributes from source object to the current object
    this->gf = srcObj.gf;

    cout<<"Copy CTOR called"<<endl;
}
```

Bad practice



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

class Student
{
private:
    string gf;

public:
    int id;
    string name;
    int age;
    string gender;

    // Default CTOR: assign garbage value
    Student(){
        cout<< "Default ctor called" <<endl;
    }

    // Parameterized CTOR: assign sensible value
    Student(int _id, string _name, int _age, string _gender, string _gf){
        id = _id;
        name = _name;
        age = _age;
        gender = _gender;
        gf = _gf;
        cout<< "Parameterized ctor called for " << name <<endl;
    }

    // Own copy CTOR: the source 's1' is coping into the destination 's2'
    Student(const Student &srcObj){
        this->id = srcObj.id;
        this->name = srcObj.name;
        this->age = srcObj.age;
        this->gender = srcObj.gender;
        this->gf = srcObj.gf;
        cout<<"Copy CTOR called"<<endl;
    }
};
```

```cpp
int main(){
    Student s1(104,"Love",25,"Male","Lovely");
    Student s2=s1;

    cout<< s1.name <<endl;
    cout<< s2.name <<endl;
    return 0;
}
```

Complete Code

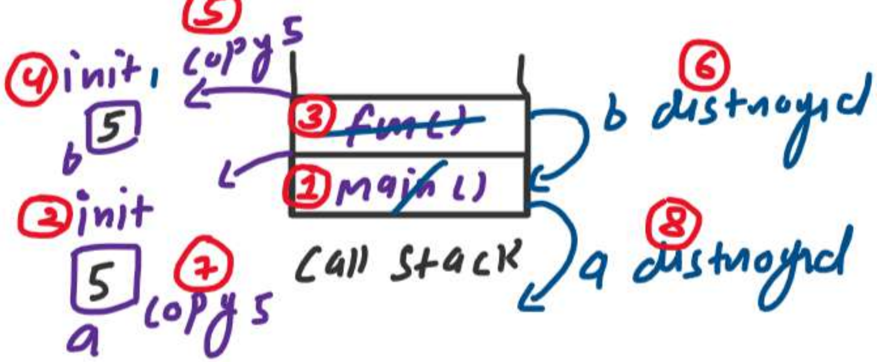
Output:

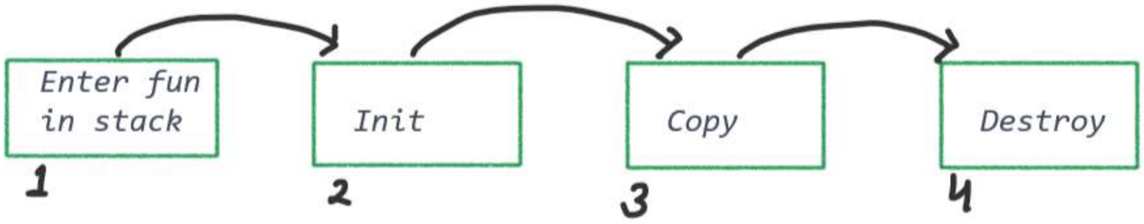
Parameterized ctor called for Love
Copy CTOR called
Love
Love

📂 2: Life cycle of an object

Life cycle of a variable

```
int main(){
    int a;
    fun();
    a=5;
    return 0;
}

void fun(){
    int b;
    b=5;
    return;
}
```





Life cycle of an object

```
int main ( ) {

    Student S1 ( data -....);   ← init S1 obj ①
    - - - - ..
    - - - - .
    - - - . ..
    return 0 ;                  ← S1 will be destroyed ②
}
```

```cpp
// 📂 Life cycle of an object
#include<iostream>
using namespace std;

class Student
{
public:
    int age;

    Student(int age){
        this->age = age;
        cout<<age<<endl;
    }
};

int main(){
    Student s1(25);
    return 0;
}
```

25

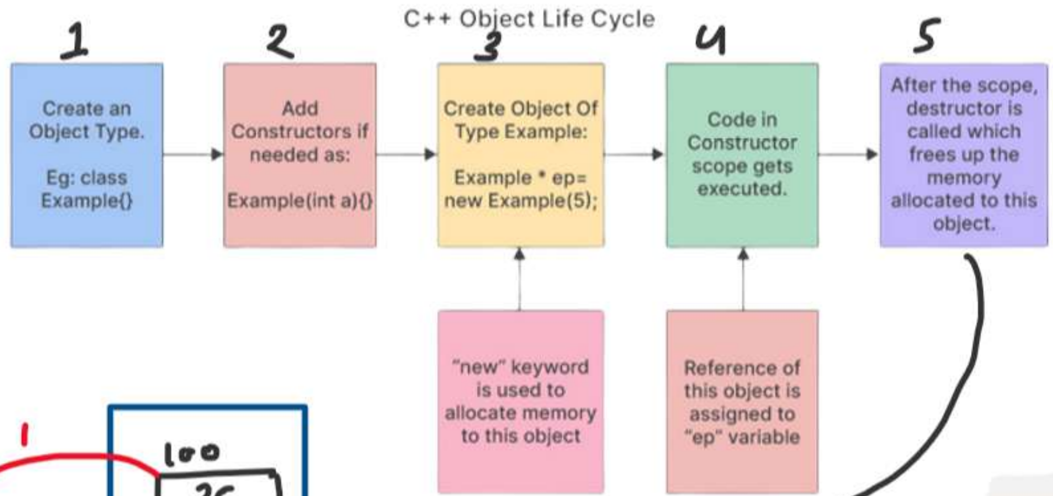
S1

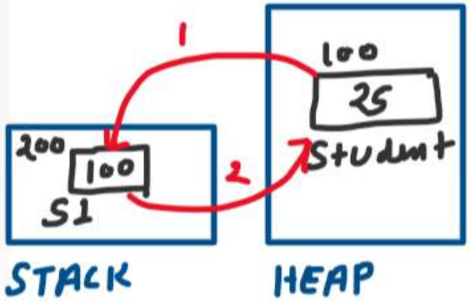
stack Mo

Output

25

```cpp
// 📁 Life cycle of an object
#include<iostream>
using namespace std;

class Student
{
public:
    int age;

    Student(int age){
        this->age = age;
        cout<<age<<endl;
    }
};

int main(){
    Student *s1 = new Student(25);
    return 0;
}
```





Note

If we do not write DTOR, compiler will take the responsibility to add a default destructor publicly.

3: Destructor in C++

PROGRAM 1

```cpp
// Destructor in C++
#include<iostream>
using namespace std;

class Student
{
public:
    int age;

    Student(){
        cout<<"Default CTOR Called"<<endl;
    }

    Student(int age){
        this->age = age;
        cout<<age<<endl;
    }

    // Default DTOR
    ~Student(){
        cout<<"Student DTOR Called"<<endl;
    }
};

int main(){
    {
        Student s1(25);
    }
    Student s2;
    return 0;
}
```

scope

In this case, class does not contain dynamic object, so we do not need to write DTOR by itself.

Jab object apna kam complete kar lega to ek default destructor(DTOR) call hoga jo object ko bhi destroyed kar dega

OUTPUT:

4 25
5 Student DTOR Called
7 Default CTOR Called
Student DTOR Called 8

S1 and S2 ARE NOT Dynamic object

Why we need of destructor?

A destructor is called automatically when the object goes out of scope or is explicitly deleted.

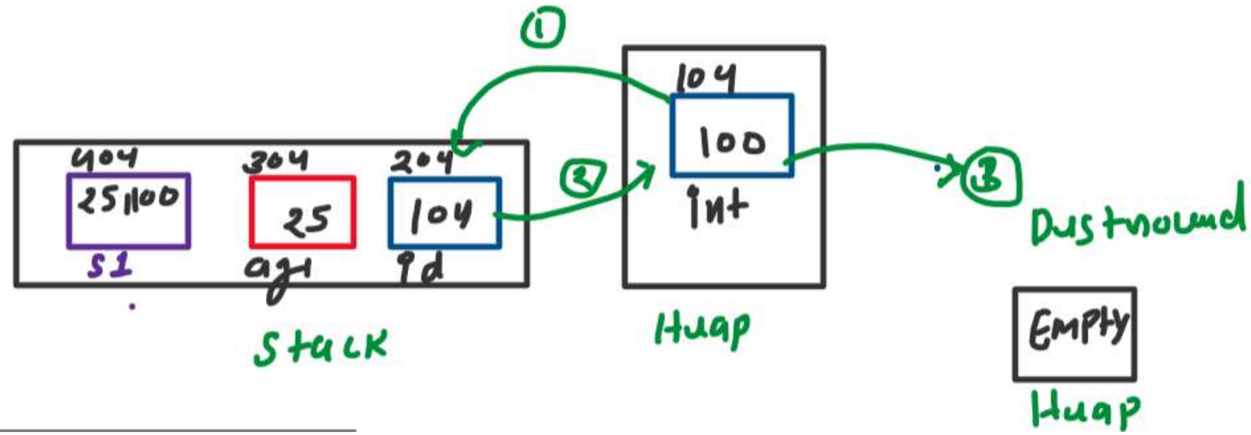
It's not mandatory to write a destructor. In majority of the cases, the compiler takes care of this for you.

However, ***when a class contains dynamic object,*** *it is mandatory to write a destructor function to release memory before the class instance is destroyed.*

PROGRAM 2

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

class Student
{
public:
    int age;
    int *id;    // 1 -> Dynamic var

    Student(int age, int id){
        this->age = age;
        this->id = new int(id);    // 2
        cout<<age<<" "<<id<<endl;
    }

    // Own DTOR
    ~Student(){
        cout<<"Student DTOR Called"<<endl;
        delete id;    // 3
    }
};

int main(){
    Student s1(25, 100);    // -> Non-dynamic obj
    return 0;
}
```

This must be done to avoid memory leak.

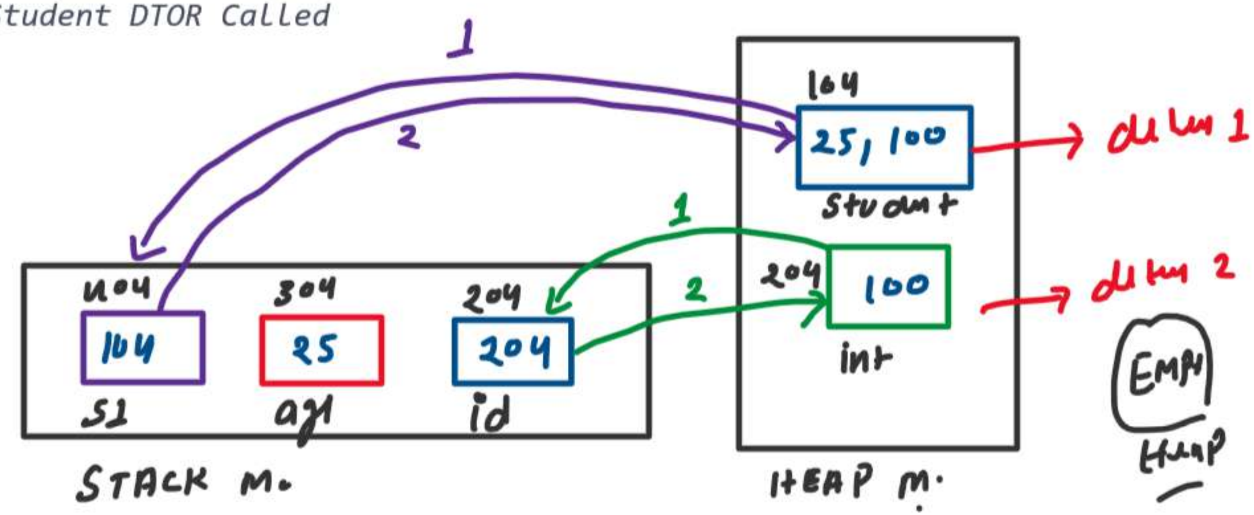


Program: 3

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

class Student
{
public:
    int age;
    int *id;   // → Dynamic var

    Student(int age, int id){
        this->age = age;
        this->id = new int(id);
        cout<<age<<" "<<id<<endl;
    }

    // Own DTOR
    ~Student(){
        cout<<"Student DTOR Called"<<endl;
        delete id;
        cout<<"Student DTOR Called"<<endl;
    }
};

int main(){
    Student *s1 = new Student(25, 100);
    delete s1;   // → Dynamic obj.
    return 0;
}
```

OUTPUT:

25 100
Student DTOR Called
Student DTOR Called



First delete S1 and second delete id

📂 4: Getter and setter methods in C++

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

class Student
{
private:
    string gf;
public:
    int age;
    int *id;

    // CTOR
    Student(int age, int id){
        this->age = age;
        this->id = new int(id);
        cout<<age<<" "<<id<<endl;
    }

    // Setter method
    void setGFName(string gf){
        this->gf = gf;
    }

    // Getter method
    string getGfName(){
        return gf;
    }

    // DTOR
    ~Student(){
        cout<<"Student DTOR Called"<<endl;
        delete id;
    }
};
```

```cpp

int main(){
    Student *s1 = new Student(25, 100);

    s1->setGFName("Lovely");

    cout<<s1->getGfName()<<endl;

    delete s1;
    return 0;
}
```

OUTPUT:

25 100
Lovely
Student DTOR Called

📂 5: Abstraction (One Pillar of OOPS)

What is abstraction?
Abstraction provides the ability to internal hide details, allowing for simpler representations of objects.
***In short,** we don't know for background implantation. Only we want to use everything.*

Jeevan me agar abstraction ho to jeevan aasan ban jayega jaise

🚓 *Car hai to usko drive karne se matlb hai only*

📱 *Phone hai to uska use karne se matlb hai only*

Abstraction three tarke se kiya ja skta hai in C++
1. Encapsulation:
it is a way to implement the abstraction by building of data and method.

2. Inheritance:
its also another way to implement the abstraction by inheriting the properties and characteristics of the super or derived class.

3. Polymorphism:
its a third way of the abstraction. In this case, we found many forms of one things.

📂 5.1: Encapsulation

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

class Student
{
private:
    string gf;
public:
    int age;
    int *id;

    // Own CTOR
    Student(int age, int id){
        this->age = age;
        this->id = new int(id);
        cout<<age<<" "<<id<<endl;
    }

    // Setter method
    void setGFName(string gf){
        this->gf = gf;
    }

    // Getter method
    string getGfName(){
        return gf;
    }

    // Own DTOR
    ~Student(){
        cout<<"Student DTOR Called"<<endl;
        delete id;
    }
};
```

```cpp

int main(){
    Student *s1 = new Student(25, 100);

    s1->setGFName("Lovely");

    cout<<s1->getGfName()<<endl;

    delete s1;
    return 0;
}
```

What is encapsulation?
It is a way to implement the abstraction by building of data and method.

In short, *encapsulation is nothing special. it's just a class.*

Why use of encapsulation?
1. Easy to handle
2. Protect integrity (Security): Control/How class data is modified
3. Maintainability

Note: *Security feature ko samjhne ke liye **'friend keyword'** samjhana bahut zaroori hai*

Protect integrity:
Authentication, *Who can access the GF name?*
✅ *Mummy, Papa, and Me*

OUTPUT:
25 100
Lovely
Student DTOR Called

5.1.1: Perfect Encapsulation

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

class Student
{
private:
    // Private data member only
    string gf;

public:
    // Setter method
    void setGFName(string gf){
        this->gf = gf;
    }

    // Getter method
    string getGfName(){
        return gf;
    }
};
```

```cpp
int main(){
    Student *s1 = new Student();

    // Set gf name through setGFName
    s1->setGFName("Lovely");

    // Get gf name through getGfName
    cout<<s1->getGfName()<<endl;

    delete s1;
    return 0;
}
```

What is Perfect encapsulation?

When two rules are followed as

1. *All data members are private.*
2. *All private data members are used through getter and setter methods.*

OUTPUT:

✔ Lovely

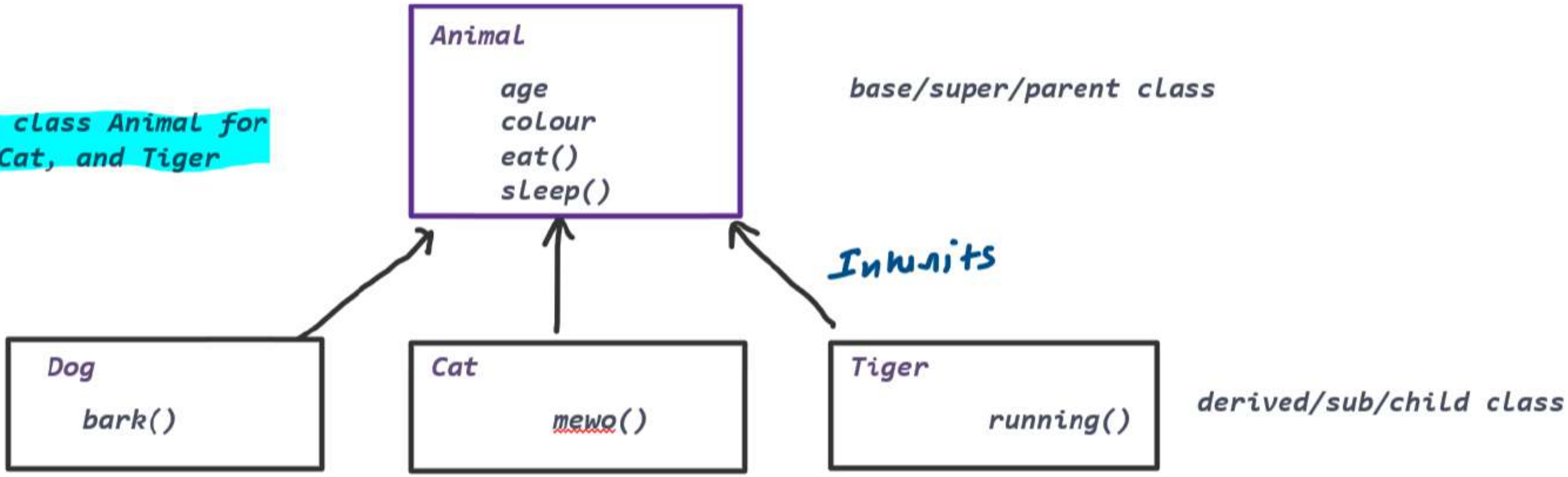
5.2: Inheritance

What is the inheritance in C++:
its a way to implement the abstraction by inheriting the properties and characteristics of the super or derived class.

1. *It allows us to create a new class (**derived/sub/child class**) from an existing class (**base/super/parent class**).*
2. *The derived class inherits the features from the base class and can have additional features of its own.*

Example:

Common features of Base class Animal for all derived class Dog, Cat, and Tiger



Syntax:

```
class Child class name : mode of in heritance Parent class name
{
   ..
};
```

↳ public/private/protected

Access Modifiers in C++: public, private, or protected

Private:
Members of base class are not accessible by derived class.
It can be only accessible for class itself. And private data can't inherit.

Protected:
Members of base class are accessible for both by derived class and class itself.

Public:
Members of base class are accessible for each derived class and class itself.
It does not provide any security.

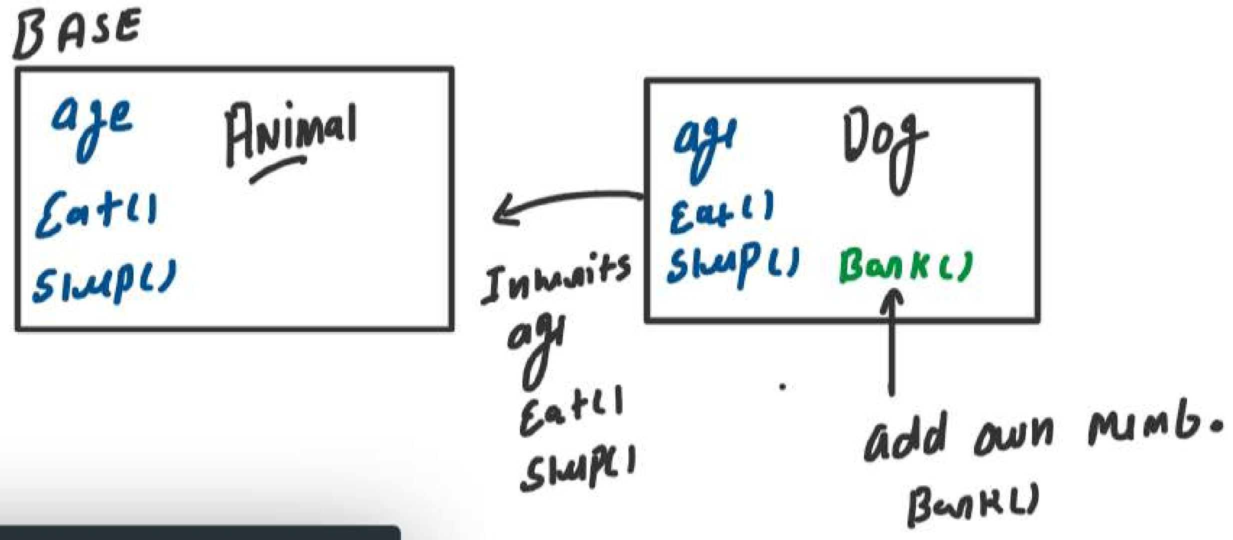
Access Modifiers in C++

Modifiers	Own Class	Derived Class	main()
Public	Yes	Yes	Yes
Private	Yes	No	No
Protected	Yes	Yes	No

Example:01

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

// Base Class Animal
class Animal
{
    public:
        int age;
        string colour;

        void eat(){
            cout<<"Eating"<<endl;
        }

        void sleep(){
            cout<<"Sleeping"<<endl;
        }
};

// Derived Class Dog
class Dog: public Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};
```

```cpp
int main(){
    Dog dogObj;
    dogObj.age = 8;
    dogObj.colour = "Black";
    dogObj.eat();
    dogObj.bark();
    cout<<dogObj.age<<" "<<dogObj.colour<<endl;
    return 0;
}
```



OUTPUT:
Eating
Barking
8 Black

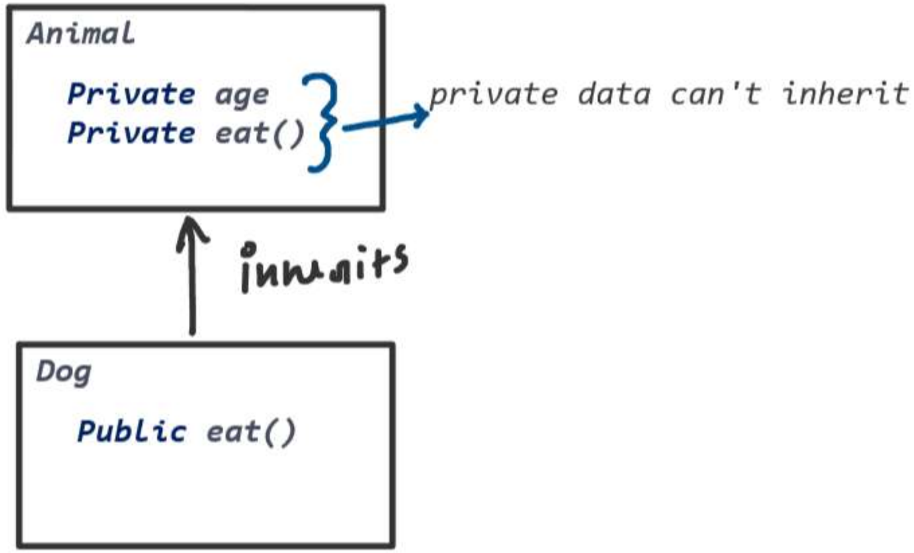
5.2.1 Mode of inheritance table

	Derived Class	Derived Class	Derived Class
Base Class	Private Mode	Protected Mode	Public Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

```cpp
// Case 01: Private member inherits as private mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    private:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: private Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

output

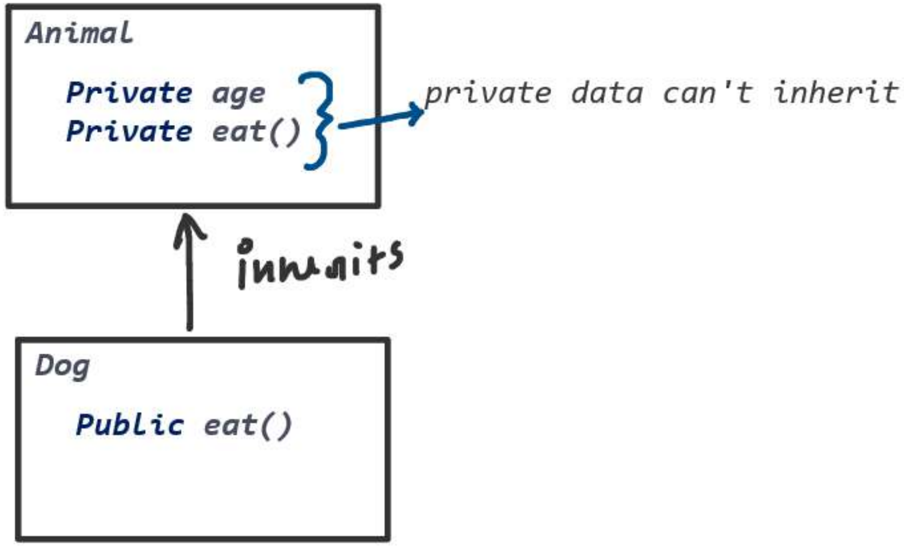
Barking



```cpp
// Case 02: Private member inherits as protected mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    private:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: protected Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

output

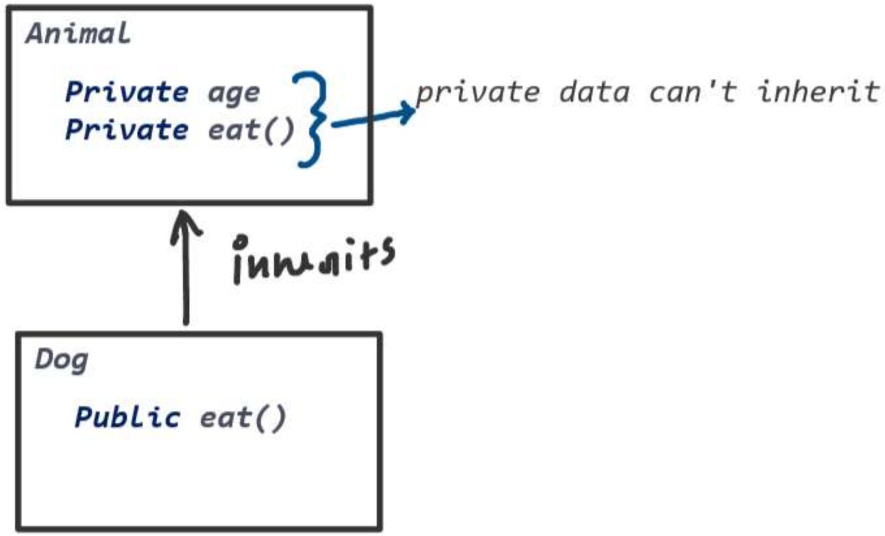
Barking



```cpp
// Case 03: Private member inherits as public mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    private:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: public Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

output

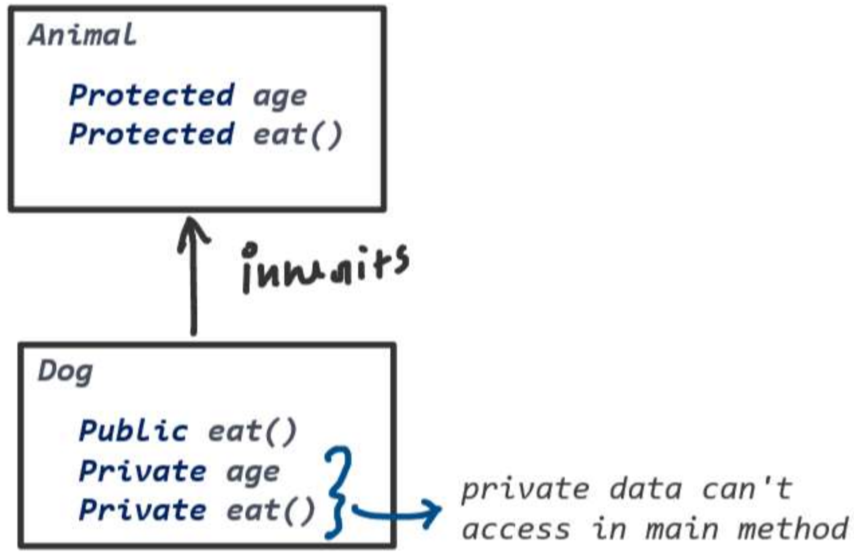
Barking



```cpp
// Case 04: Protected member inherits as private mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    protected:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: private Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

output

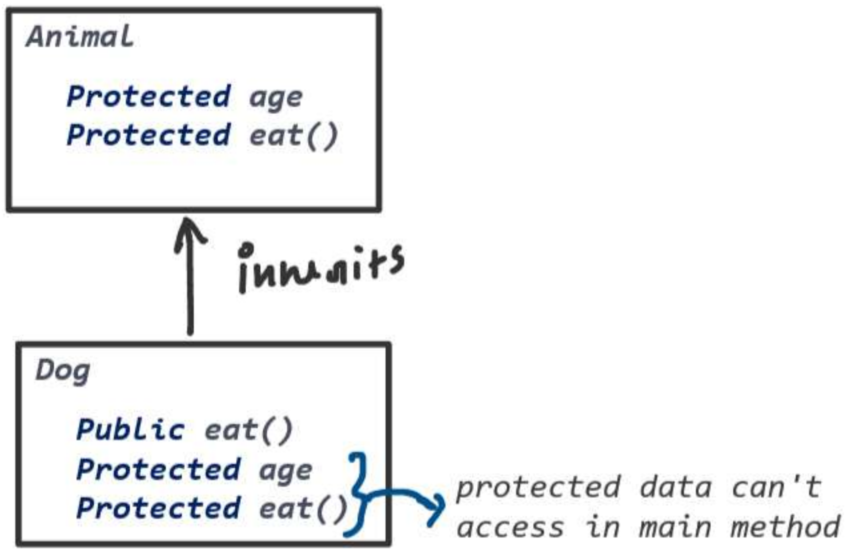
Barking



```cpp
// Case 05: Protected member inherits as protected mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    protected:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: protected Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

output

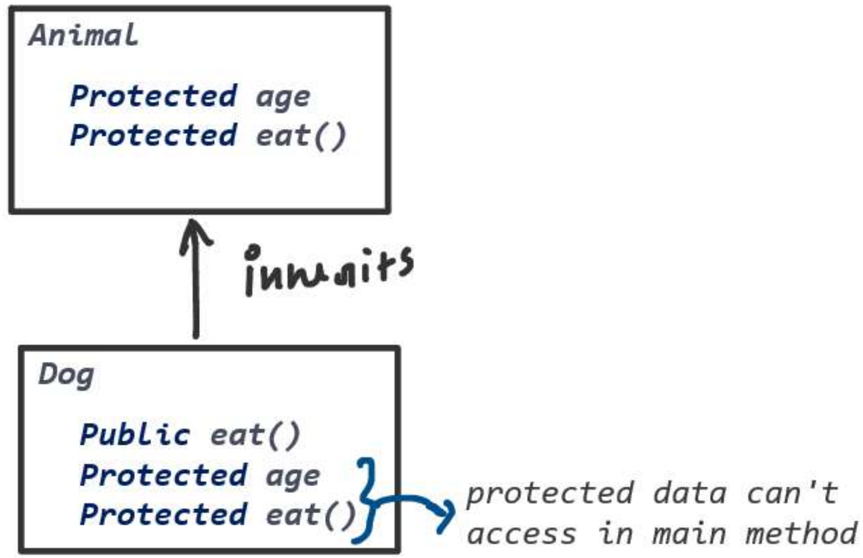
Barking



```cpp
// Case 06: Protected member inherits as public mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    protected:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: public Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

Output

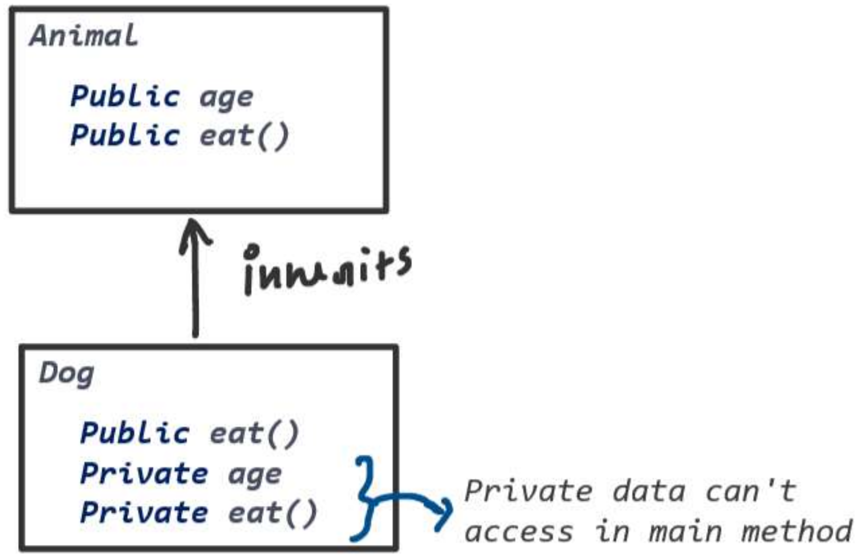
Barking



```cpp
// Case 07: Public member inherits as private mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    public:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: private Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

output

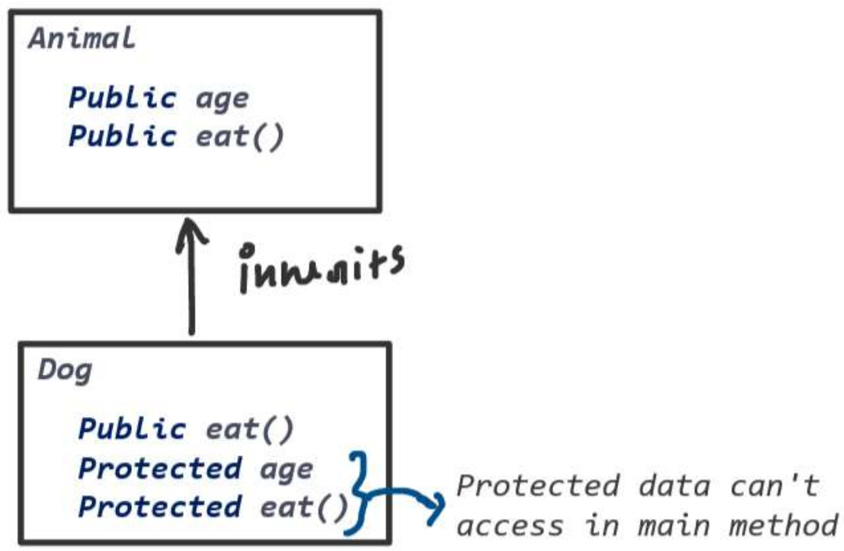
Barking



```cpp
// Case 08: Public member inherits as protected mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    public:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: protected Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    return 0;
}
```

Output

Barking

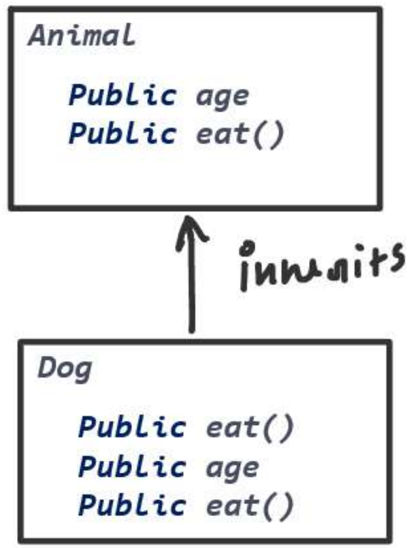


```cpp
// Case 09: Public member inherits as public mode
#include<iostream>
using namespace std;

// Base Class Animal
class Animal
{
    public:
        int age;

        void eat(){
            cout<<"Eating"<<endl;
        }
};

// Derived Class Dog
class Dog: public Animal
{
    public:
        void bark(){
            cout<<"Barking"<<endl;
        }
};

int main(){
    Dog dogObj;
    dogObj.bark();
    dogObj.age = 8;
    dogObj.eat();
    return 0;
}
```

age $\boxed{8}$

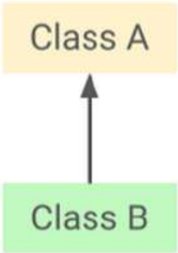
<u>output</u>

Barking
Eating

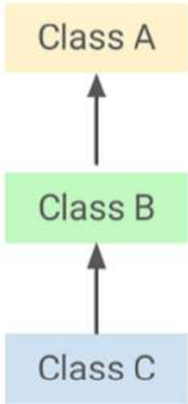


Public data can access in main method

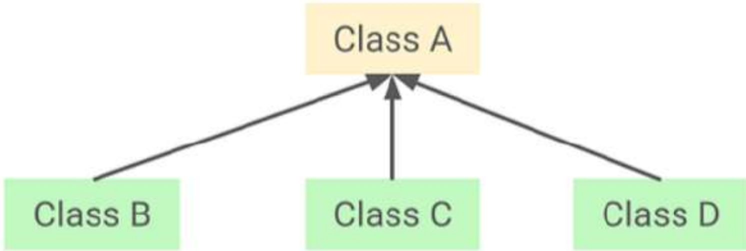
📂 5.2.2 Type of inheritance



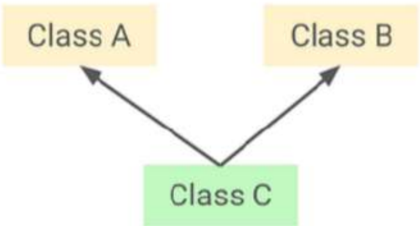
Type 01: Single



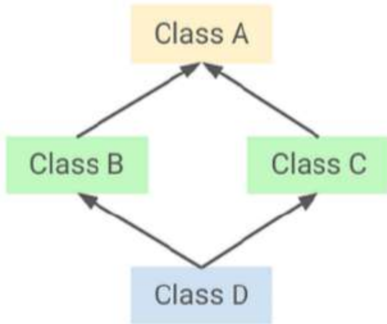
Type 02: Multilevel



Type 03: Hierarchical

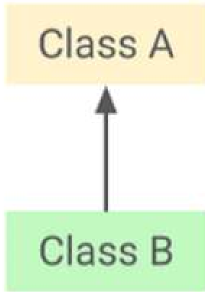


Type 04: Multiple



Type 05: Diamond Problem (Hybrid inheritance)

```cpp
// 📁 5.2.2.1 Single inheritance program
#include<iostream>
using namespace std;

// Base Class A
class A
{
    public:
        int id;

        void funA(){
            cout<<"FunA called"<<endl;
        }
};

// Derived Class B
class B: public A
{
    public:
        void funB(){
            cout<<"FunB called"<<endl;
        }
};

int main(){
    B BObj;
    BObj.funA();
    BObj.id = 8;
    BObj.funB();
    return 0;
}
```

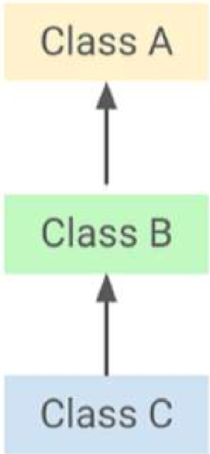


Type 01: Single

Output:
FunA called
FunB called

```cpp
// 📁 5.2.2.2 Multilevel inheritance program
#include<iostream>
using namespace std;

// Base Class A for B
class A
{
    public:
        int id;

        void funA(){
            cout<<"FunA called"<<endl;
        }
};

// Derived Class B for A and Base class B for C
class B: public A
{
    public:
        void funB(){
            cout<<"FunB called"<<endl;
        }
};

// Derived Class C for B
class C: public B
{
    public:
        void funC(){
            cout<<"FunC called"<<endl;
        }
};
```

```cpp
int main(){
    B BObj;
    BObj.funA();
    BObj.funB();
    BObj.id = 8;

    C CObj;
    CObj.funA();
    CObj.funB();
    CObj.funC();
    CObj.id = 10;
    return 0;
}
```



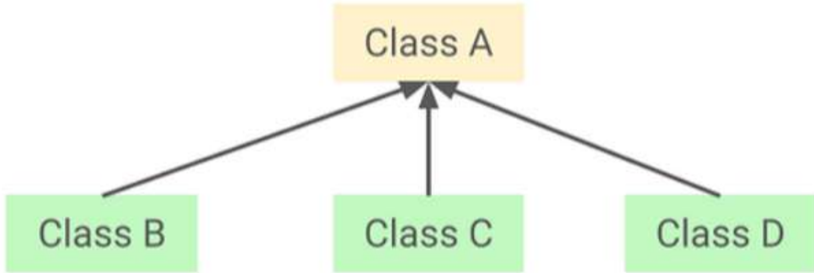
Type 02: Multilevel

Output:

```
FunA called
FunB called
FunA called
FunB called
FunC called
```

```cpp
// 📁 5.2.2.3 Hierarchical inheritance program
#include<iostream>
using namespace std;

// Base Class A for B, C, and D
class A
{
    public:
        int id;

        void funA(){
            cout<<"FunA called"<<endl;
        }
};

// Derived Class B for A
class B: public A
{
    public:
        void funB(){
            cout<<"FunB called"<<endl;
        }
};

// Derived Class C for A
class C: public A
{
    public:
        void funC(){
            cout<<"FunC called"<<endl;
        }
};

// Derived Class D for A
class D: public A
{
    public:
        void funD(){
            cout<<"FunD called"<<endl;
        }
};
```

```cpp

int main(){
    B BObj;
    BObj.funA();
    BObj.funB();
    BObj.id = 8;

    C CObj;
    CObj.funA();
    CObj.funC();
    CObj.id = 10;

    D DObj;
    DObj.funA();
    DObj.funD();
    DObj.id = 12;
    return 0;
}
```



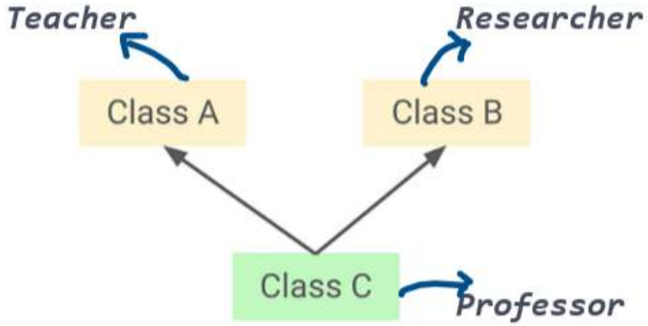
Type 03: Hierarchical

Output:

FunA called
FunB called
FunA called
FunC called
FunA called
FunD called

```cpp
// 📁 5.2.2.4 Multiple inheritance program
#include<iostream>
using namespace std;

// Base Class A as Teacher
class Teacher
{
    public:
        void teach(){
            cout<<"Teaching"<<endl;
        }
};

// Base Class B as Researcher
class Researcher
{
    public:
        void research(){
            cout<<"Researching"<<endl;
        }
};

// Derived Class C as Professor
class Professor: public Teacher, public Researcher
{
    public:
        void bore(){
            cout<<"Boring"<<endl;
        }
};
```

```cpp
int main(){
    Professor PObje;
    PObje.bore();
    PObje.teach();
    PObje.research();
    return 0;
}
```

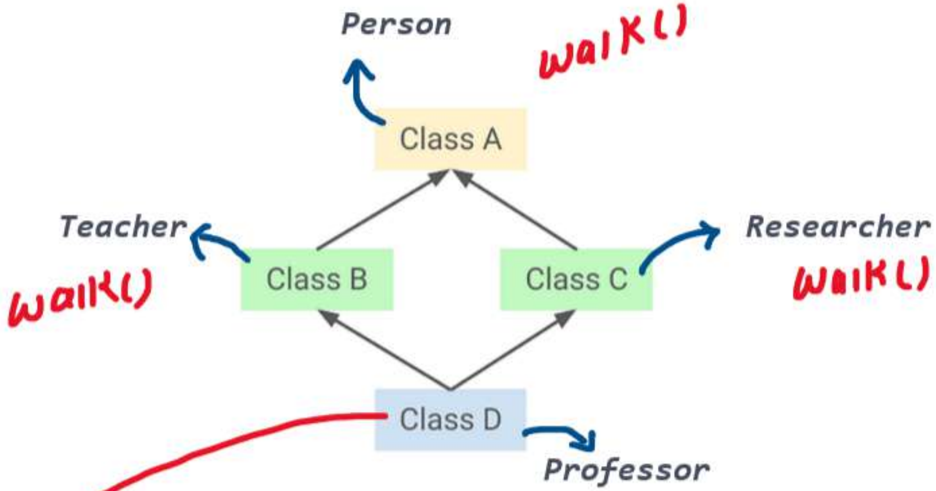


Type 04: Multiple

Output:
Boring
Teaching
Researching

```cpp
// 📁 5.2.2.5 Diamond Problem (Hybrid inheritance)
#include<iostream>
using namespace std;

// Base Class A as Person for Teacher and Researcher
class Person
{
    public:
        void walk(){
            cout<<"Walking"<<endl;
        }
};

// Derived class B as Teacher for Person and  Base Class Teacher for Professor
class Teacher: public Person
{
    public:
        void teach(){
            cout<<"Teaching"<<endl;
        }
};

// Derived class C as Researcher for Person and  Base Class Researcher for Professor
class Researcher: public Person
{
    public:
        void research(){
            cout<<"Researching"<<endl;
        }
};

// Derived Class D as Professor for Teacher and Researcher
class Professor: public Teacher, public Researcher
{
    public:
        void bore(){
            cout<<"Boring"<<endl;
        }
};

int main(){
    Professor PObje;
    PObje.walk(); // error: request for member 'walk' is ambiguous
    return 0;
}
```

ERROR



Type 05: Diamond Problem (Hybrid inheritance)

Ambiguous Problem Error: *Now Compiler has confused ki Professor ko konsa walk doo Teacher se ya fir Researcher se*

```cpp
// 📂 5.2.2.5.1 Diamond Problem (Hybrid inheritance) with Scope resolution
#include<iostream>
using namespace std;

// Base Class A as Person for Teacher and Researcher
class Person
{
    public:
        void walk(){
            cout<<"Walking"<<endl;
        }
};

// Derived class B as Teacher for Person and  Base Class Teacher for Professor
class Teacher: public Person
{
    public:
        void teach(){
            cout<<"Teaching"<<endl;
        }
};

// Derived class C as Researcher for Person and  Base Class Researcher for Professor
class Researcher: public Person
{
    public:
        void research(){
            cout<<"Researching"<<endl;
        }
};
```

Ambiguous Problem Error Solution 01

```cpp
// Derived Class D as Professor for Teacher and Researcher
class Professor: public Teacher, public Researcher
{
    public:
        void bore(){
            cout<<"Boring"<<endl;
        }
};

int main(){
    Professor PObj;
    // Diamond Problem
    // Solution 1: Scope Resolution
    PObj.Teacher::walk();
    PObj.Researcher::walk();
    return 0;
}
```

Output:
Walking
Walking

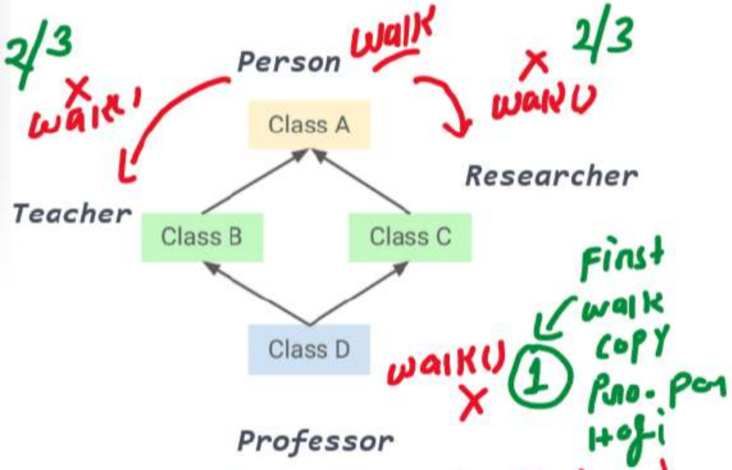
Not Good Sol.n

In this solution, compiler is giving Teacher walk() and Researcher walk() individually

```cpp
// 📁 5.2.2.5.2 Diamond Problem Solution Using Virtual
#include<iostream>
using namespace std;

class Person
{
    public:
        void walk(){
            cout<<"Walking"<<endl;
        }
};

class Teacher: virtual public Person
{
    public:
        void teach(){
            cout<<"Teaching"<<endl;
        }
};

class Researcher: virtual public Person
{
    public:
        void research(){
            cout<<"Researching"<<endl;
        }
};

class Professor: public Teacher, public Researcher
{
    public:
        void bore(){
            cout<<"Boring"<<endl;
        }
};
```

```cpp
int main(){
    Professor PObje;
    // Diamond Problem
    // Solution 2: using virtual
    PObje.walk();
    return 0;
}
```

Ambiguous Problem Error **Solution 02**



Output:
Walking

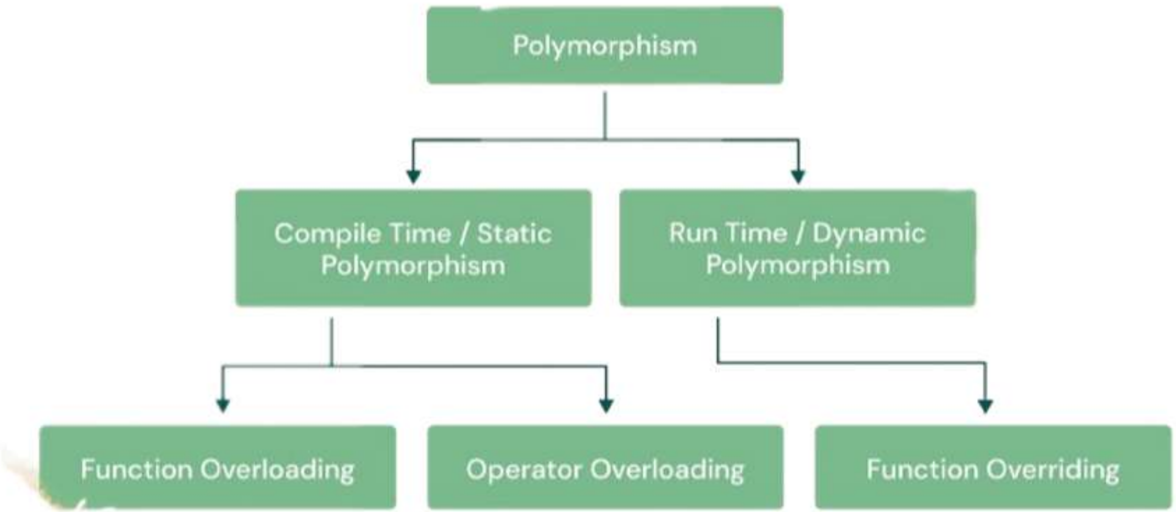
Compile Time par walk() ki copy nahi milti Hai

Teacher, Researcher, Prof. – In sabko Runtime par copy assign ki jati Hai.

📂 5.3 Polymorphism

What is polymorphism:
This is a way of the abstraction. In this case, we found many forms of one things.

Types of polymorphism:



1. Compile time:

```cpp
//📁 (I.) Function Overloading Program
#include <iostream>
using namespace std;

// First fn
int add(int a, int b)
{
    return a + b;
}

// Second fn: different types of parameters from first
double add(double a, double b)
{
    return a + b;
}

// Third fn: different number of parameters from first
int add(int a, int b, int c)
{
    return a + b + c;
}

int main(){
    cout<< add(5, 10) << endl;
    cout<< add(5.5, 10.5) << endl;
    cout<< add(5,10, 15) << endl;
    return 0;
}
```

(I.) Function overloading:

Two or more function can have same name but different parameters.

Require each redefinition of a function to use a different function signature that is:

a.) Different types of parameters
b.) Or sequence of parameters
c.) Or number of parameters

OUTPUT:
15 5+10 = 15
16 5+10+1 = 16
30 5+10+15=30

(II.) Operator overloading:

Ex:1 +

```cpp
// (II.) Operator Overloading Program
#include <iostream>
using namespace std;

class Vector
{
    private:
        int x, y;

    public:
        // Init list CTOR
        Vector(int x, int y): x(x), y(y){}

        // Simple member fn
        void display(){
            cout<<"x: "<<x<<" y: "<<y<<endl;
        }

        // Operator overlading
        void operator+(const Vector &src)
        {
            // this would point to v1
            // src would point to v2
            this->x += src.x;
            this->y += src.y;
        }
};
```

```cpp
int main(){
    Vector v1(2, 3);
    Vector v2(4, 5);

    cout<<"Before operator overloading"<<endl;
    v1.display();
    v2.display();

    cout<<"After operator overloading"<<endl;
    v1 + v2;
    // Additional ans (v1+v2) should be stored in v1
    v1.display();
    v2.display();

    return 0;
}
```

$$\underset{V_1}{\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}} + \underset{V_2}{\begin{bmatrix} x_2 \\ y_2 \end{bmatrix}} = \underset{V_1+V_2}{\begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \end{bmatrix}}$$

$V_1 + V_2 \Rightarrow$ src $= V_2$, dest $= V_1$ $\Rightarrow$ means $\begin{bmatrix} x_1+x_2 \\ y_1+y_2 \end{bmatrix}$

$V_1 = V_1 + V_2$

Ex

$$\underset{V_1}{\begin{bmatrix} 2 \\ 3 \end{bmatrix}} + \underset{V_2}{\begin{bmatrix} 4 \\ 5 \end{bmatrix}} = \underset{V_1 = V_1+V_2}{\begin{bmatrix} 6 \\ 8 \end{bmatrix}}$$

OUTPUT:

Before operator overloading
x: 2 y: 3 — V1
x: 4 y: 5 — V2
After operator overloading
x: 6 y: 8 — V1
x: 4 y: 5 — V2

Addition operation (+)

Destination → V_1 + V_2 → source

```
void operator+ (const Vector &src)
{
```

this $\xrightarrow{\text{points}}$ V1

src $\xrightarrow{\text{points}}$ V2

```
    this->x = this->x + src.x;
    this->y = this->y + src.y;
}
```

Subtract operation (-)

Destination

V_2 − V_1 → source

void operator-(const Vector &src)

{

this $\xrightarrow{\text{points}}$ V_2

src $\xrightarrow{\text{points}}$ V_1

this->x = this->x - src.x;

this->y = this->y - src.y;

}

Ex:2 (−)

```cpp
//📁 (II.) Operator Overloading Program
#include <iostream>
using namespace std;

class Vector
{
    private:
        int x, y;

    public:
        // Init list CTOR
        Vector(int x, int y): x(x), y(y){}

        // Simple member fn
        void display(){
            cout<<"x: "<<x<<" y: "<<y<<endl;
        }

        // Operator overlading
        void operator-(const Vector &src)
        {
            // this would point to v2
            // src would point to v1
            this->x -= src.x;
            this->y -= src.y;
        }
};
```

```cpp
int main(){
    Vector v1(2, 3);
    Vector v2(4, 5);

    cout<<"Before operator overloading"<<endl;
    v1.display();
    v2.display();

    cout<<"After operator overloading"<<endl;
    v2 - v1;
    // Subtraction ans (v2-v1) should be stored in v2
    v1.display();
    v2.display();

    return 0;
}
```

$$\underset{V_2}{\begin{bmatrix} 4 \\ 5 \end{bmatrix}} - \underset{V_1}{\begin{bmatrix} 2 \\ 3 \end{bmatrix}} = \underset{V_2 = V_2 - V_1}{\begin{bmatrix} 2 \\ 2 \end{bmatrix}}$$

OUTPUT:

```
Before operator overloading
x: 2 y: 3 — V1
x: 4 y: 5 — V2
After operator overloading
x: 2 y: 3 — V1
x: 2 y: 2 — V2
```

Which operators overload in C++?

You can overload the following operators in C++

1. Unary arithmetic operators: **+, -, ++, --**
2. Binary arithmetic operators: **+, -, *, /, %**
3. Assignment operators: **=, +=, *=, /=, -=, %=**
4. Bitwise operators: **&, |, <<, >>, ~, ^**
5. Function call operator: **()**

You cannot overload the following operators in C++

1. **size of,**
2. **.** (member selection)
3. **?:** (conditional)
4. **::** (scope resolution)
5. **new**
6. **delete**