

25/10/2023

Object Oriented Programming Class 02

1: Copy Constructor

```
1 #include<iostream>
2 using namespace std;
3
4 class Student
5 {
6 private:
7     string gf;
8
9     void chatting(){
10         cout<< "Chatting " << endl;
11     }
12
13 public:
14     int id;
15     string name;
16     int age;
17     string gender;
18
19     // Default CTOR: assign garbage value
20     Student(){
21         cout<< "Default ctor called" << endl;
22     }
23
24     // Parameterized CTOR: assign sensible value
25     Student(int _id, string _name, int _age, string _gender, string _gf){
26         id = _id;
27         name = _name;
28         age = _age;
29         gender = _gender;
30         gf = _gf;
31         cout<< "Parameterized ctor called for " << name << endl;
32     }
33
34     void study(){
35         cout<< "Studying" << endl;
36     }
37 };
```

```
1 int main(){
2
3     1 Student s1(104,"Love",25,"Male","Lovely");
4
5     2 Student s2;
6
7     3 s2 = s1;
8
9     4 cout<< s2.age << " " << s2.name << endl;
10
11     return 0;
12 }
```

- 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 jaygi

```
1 int main(){  
2  
3     Student s1(104,"Love",25,"Male","Lovely");  
4  
5     1 Student s2 = s1; // or s2(s1);  
6  
7     cout<< s2.age << " " << s2.name << endl;  
8  
9     return 0;  
10 }
```

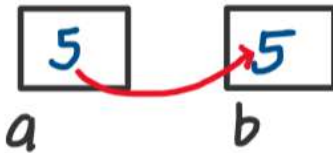
① Destination = Source, Now copied s1 into s2 (In this case, also copy constructor added by compiler)

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 integers copied
Dest. Src.

jab main



s1 copied into s2

student s1(Data);
student s2 = s1; two student copied
Dest. Src.

this points to s2

```
1 // Own copy Ctor: the source 's1' is coping into the destination 's2'
2 Student(const Student &srcObj){
3     // Copy attributes from source object to the current object
4     this->id = srcObj.id;
5     this->name = srcObj.name;
6     this->age = srcObj.age;
7     this->gender = srcObj.gender;
8     this->gf = srcObj.gf;
9     cout<<"Copy Ctor called"<<endl;
10 }
```

COPY CTOR NAME

srcObj points to s2

REFERENCE TO AN OBJECT OF CLASS s1

COPY CTOR BODY
{..}

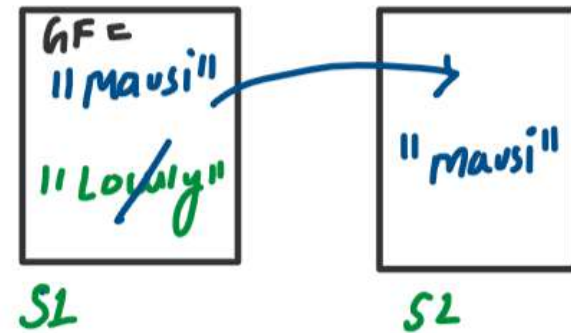
Why we need of copy constructor?

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

Why we need of const in copy constructor?

```
1 // Own copy CTOR: the source 's1' is coping into the destination 's2'
2 Student(Student &srcObj){
3
4     // Hacker
5     srcObj.gf = "Mausi 😊";
6
7     // Copy attributes from source object to the current object
8     this->gf = srcObj.gf;
9
10    cout<<"Copy CTOR called"<<endl;
11 }
```

→ Bad Practice



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

```
3
4 class Student
5 {
6 private:
7     string gf;
8
9 public:
10     int id;
11     string name;
12     int age;
13     string gender;
```

```
14
15 // Default CTOR: assign garbage value
16 Student(){
17     cout<< "Default ctor called" <<endl;
18 }
```

```
19
20 // Parameterized CTOR: assign sensible value
21 Student(int _id, string _name, int _age, string _gender, string _gf){
22     id = _id;
23     name = _name;
24     age = _age;
25     gender = _gender;
26     gf = _gf;
27     cout<< "Parameterized ctor called for " << name <<endl;
28 }
```

```
29
30 // Own copy CTOR: the source 's1' is coping into the destination 's2'
31 Student(const Student &srcObj){
32     this->id = srcObj.id;
33     this->name = srcObj.name;
34     this->age = srcObj.age;
35     this->gender = srcObj.gender;
36     this->gf = srcObj.gf;
37     cout<<"Copy CTOR called"<<endl;
38 }
39 };
```

```
1 int main(){
2     Student s1(104,"Love",25,"Male","Lovely");
3     Student s2=s1;
4
5     cout<< s1.name <<endl;
6     cout<< s2.name <<endl;
7     return 0;
8 }
```

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;
```

```
}
```

Enter fun
in stack

1

Init

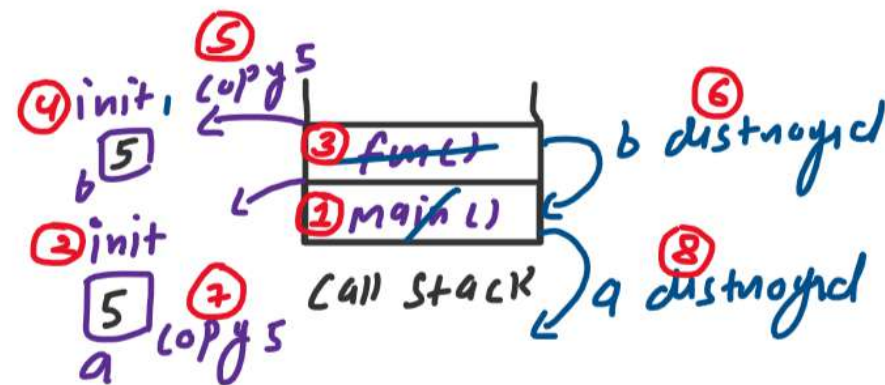
2

Copy

3

Destroy

4



Life cycle of an object

```
int main () {  
    Student s1 (data-...);  
    .....  
    return 0;  
}
```

init s1 obj ①

s1 will be destroyed ②

```
1 // Life cycle of an object  
2 #include<iostream>  
3 using namespace std;  
4  
5 class Student  
6 {  
7 public:  
8     int age;  
9  
10    Student(int age){  
11        this->age = age;  
12        cout<<age<<endl;  
13    }  
14 };  
15  
16 int main(){  
17     Student s1(25);  
18     return 0;  
19 }
```

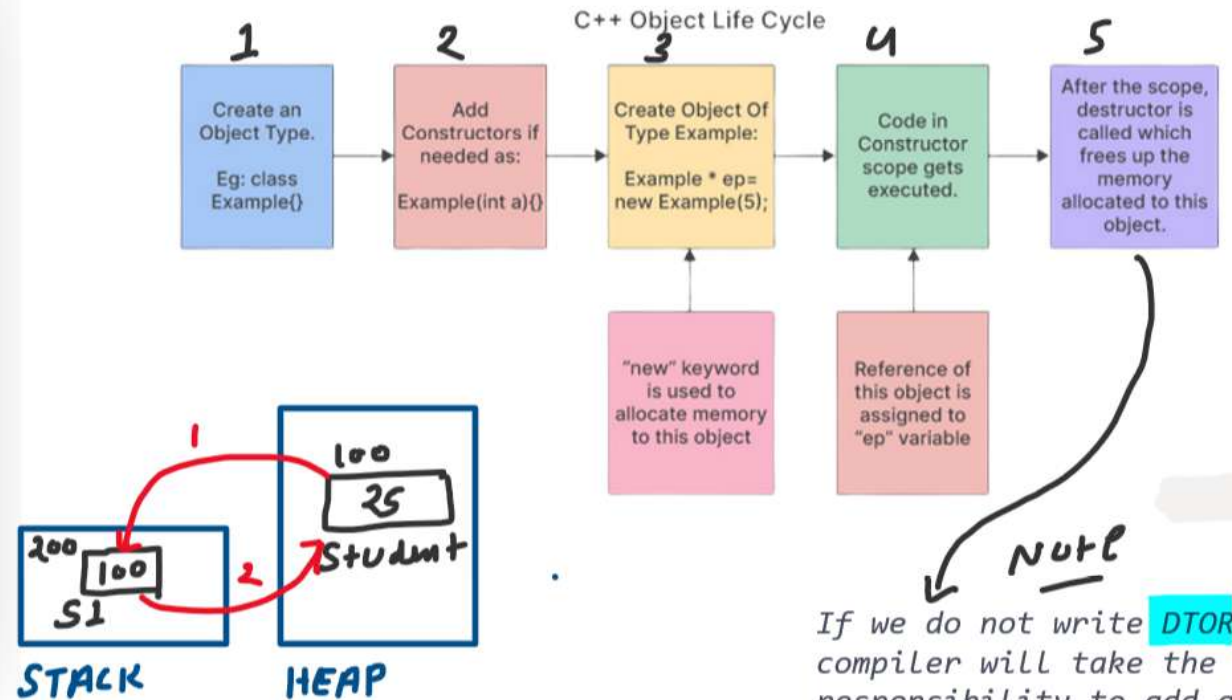
25
s1
stack m.

Output
25

```

1 // 📁 Life cycle of an object
2 #include<iostream>
3 using namespace std;
4
5 class Student 1
6 {
7 public:
8     int age;
9
10 2 Student(int age){
11     this->age = age;
12     cout<<age<<endl;
13 }
14 };
15
16 int main(){
17 3 Student *s1 = new Student(25);
18     return 0; 5
19 }

```



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

3: Destructor in C++

PROGRAM 1

```
1 // Destructure in C++
2 #include<iostream>
3 using namespace std;
4
5 class Student 1
6 {
7 public:
8     int age;
9
10    Student(){ 7
11        cout<<"Default CTOR Called"<<endl;
12    }
13
14    2 Student(int age){
15        this->age = age;
16        cout<<age<<endl; 4
17    }
18
19    // Default DTOR 5 8
20    ~Student(){
21        cout<<"Student DTOR Called"<<endl;
22    }
23 },
24
25 int main(){
26     {
27         3 Student s1(25);
28     }
29     Student s2; 6
30     return 0;
31 }
```

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.

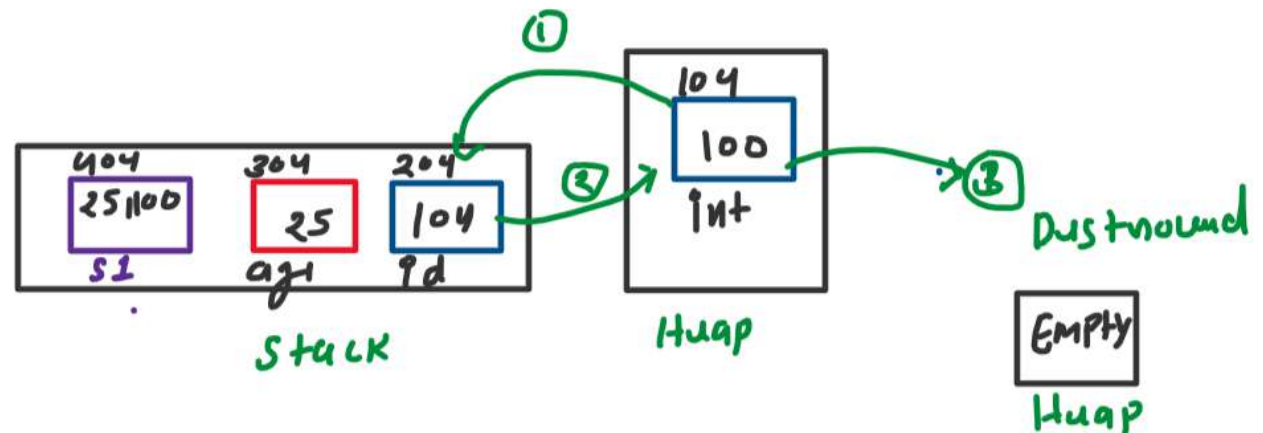
It's not mandatory to write a destructor. In majority of the cases, the compiler takes care of this for you.

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
1 #include<iostream>
2 using namespace std;
3
4 class Student
5 {
6 public:
7     int age;
8     int *id; 1 → Dynamic Var
9
10    Student(int age, int id){
11        this->age = age;
12        this->id = new int(id); 2
13        cout<<age<<" "<<id<<endl;
14    }
15
16    // Own DTOR
17    ~Student(){
18        cout<<"Student DTOR Called"<<endl;
19        delete id; 3
20    }
21 };
22
23 int main(){
24     Student s1(25, 100);
25     return 0;
26 }
```

Non-dynamic obj

This must be done to avoid memory Leak.



Program: 3

```

1 #include<iostream>
2 using namespace std;
3
4 class Student
5 {
6 public:
7     int age;
8     int *id;
9
10    Student(int age, int id){
11        this->age = age;
12        this->id = new int(id);
13        cout<<age<<" "<<id<<endl;
14    }
15
16    // Own DTOR
17    ~Student(){
18        cout<<"Student DTOR Called"<<endl;
19        delete id;
20        cout<<"Student DTOR Called"<<endl;
21    }
22 };
23
24 int main(){
25     Student *s1 = new Student(25, 100);
26     delete s1;
27     return 0;
28 }

```

dynamic U var

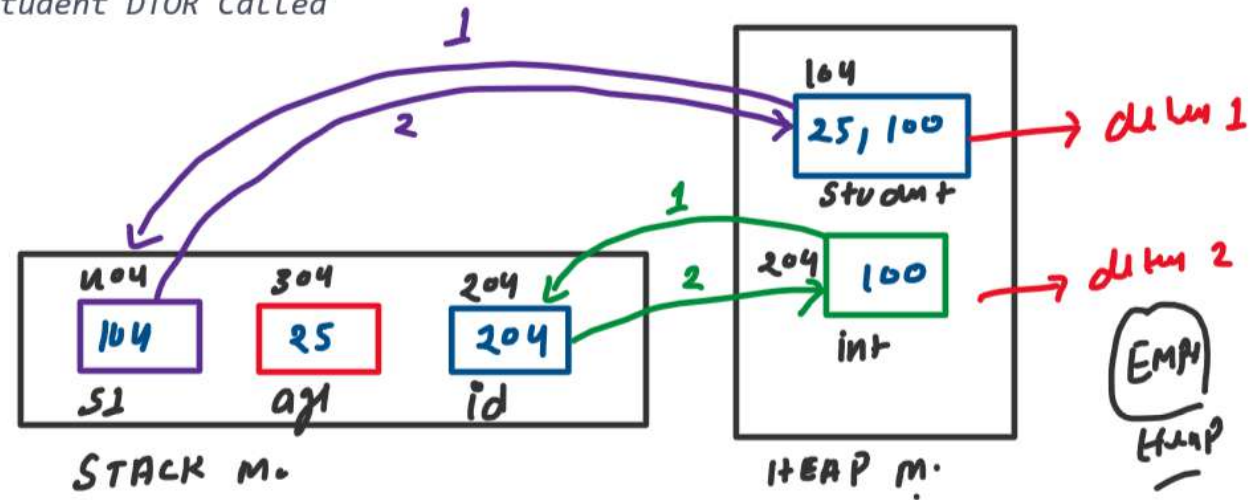
dynamic obj.

OUTPUT:

```

25 100
Student DTOR Called
Student DTOR Called

```



First delete `s1` and second delete `id`

4: Getter and setter methods in C++

```
1 #include<iostream>
2 using namespace std;
3
4 class Student
5 {
6 private:
7     string gf;
8 public:
9     int age;
10    int *id;
11
12    // CTOR
13    Student(int age, int id){
14        this->age = age;
15        this->id = new int(id);
16        cout<<age<<" "<<id<<endl;
17    }
18
19    // Setter method
20    void setGFName(string gf){
21        this->gf = gf;
22    }
23
24    // Getter method
25    string getGFName(){
26        return gf;
27    }
28
29    // DTOR
30    ~Student(){
31        cout<<"Student DTOR Called"<<endl;
32        delete id;
33    }
34 };
```

```
1
2 int main(){
3     Student *s1 = new Student(25, 100);
4
5     1 s1->setGFName("Lovely");
6
7     2 cout<<s1->getGFName()<<endl;
8
9     delete s1;
10    return 0;
11 }
```

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 javega 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

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

```
1 #include<iostream>
2 using namespace std;
3
4 class Student
5 {
6 private:
7     string gf;
8 public:
9     int age;
10    int *id;
11
12    // Own CTOR
13    Student(int age, int id){
14        this->age = age;
15        this->id = new int(id);
16        cout<<age<<" "<<id<<endl;
17    }
18
19    // Setter method
20    void setGFName(string gf){
21        this->gf = gf;
22    }
23
24    // Getter method
25    string getGFName(){
26        return gf;
27    }
28
29    // Own DTOR
30    ~Student(){
31        cout<<"Student DTOR Called"<<endl;
32        delete id;
33    }
34 };
```

```
1
2 int main(){
3     Student *s1 = new Student(25, 100);
4
5     s1->setGFName("Lovely");
6
7     cout<<s1->getGFName()<<endl;
8
9     delete s1;
10    return 0;
11 }
```

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

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.*

```
1 #include<iostream>
2 using namespace std;
3
4 class Student
5 {
6 private:
7     // Private data member only
8     string gf;
9
10 public:
11     // Setter method
12     void setGFName(string gf){
13         this->gf = gf;
14     }
15
16     // Getter method
17     string getGfName(){
18         return gf;
19     }
20 };
```

```
1 int main(){
2     Student *s1 = new Student();
3
4     // Set gf name through setGFName
5     s1->setGFName("Lovely");
6
7     // Get gf name through getGfName
8     cout<<s1->getGfName()<<endl;
9
10    delete s1;
11    return 0;
12 }
```

OUTPUT:
✓ Lovely

5.2: Inheritance

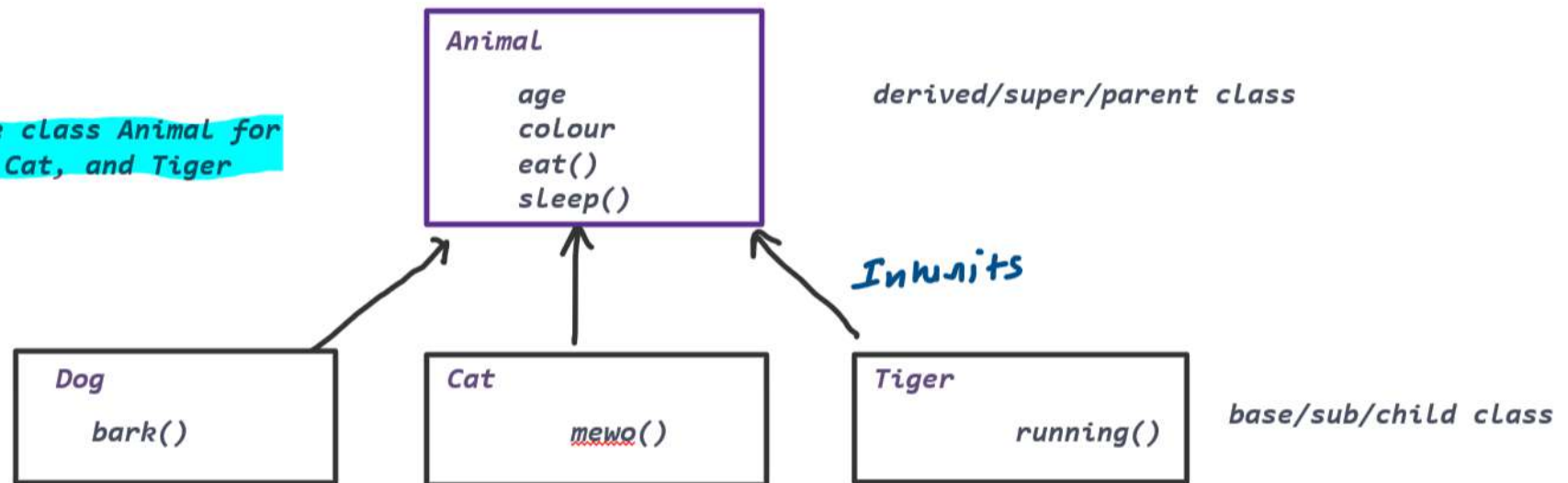
What is the inheritance in C++:

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/super/parent class**) from an existing class (**base/sub/child 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 inheritance 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++

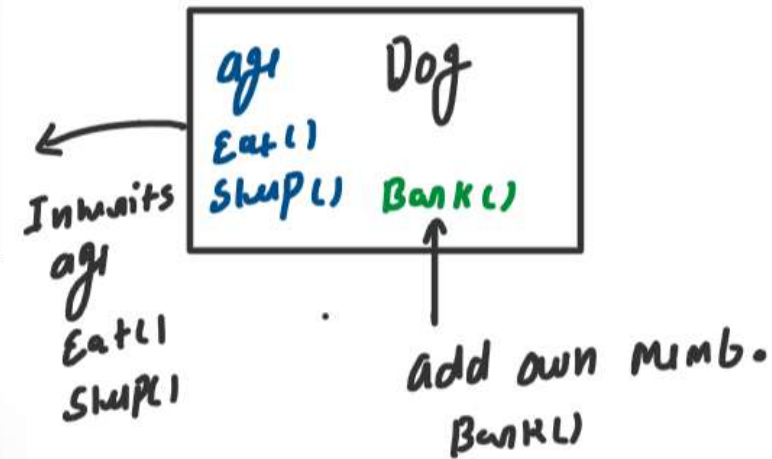
Modifiers	Own Class	Derived Class	main()
Public	Yes	Yes	Yes
Private	Yes	No	No
Protected	Yes	Yes	No

Example:01

```
1 #include<iostream>
2 using namespace std;
3
4 // Base Class Animal
5 class Animal
6 {
7     public:
8         int age;
9         string colour;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14
15         void sleep(){
16             cout<<"Sleeping"<<endl;
17         }
18 };
19
20 // Derived Class Dog
21 class Dog: public Animal
22 {
23     public:
24         void bark(){
25             cout<<"Barking"<<endl;
26         }
27 };
```

```
1 int main(){
2     Dog dogObj;
3     dogObj.age = 8;
4     dogObj.colour = "Black";
5     dogObj.eat();
6     dogObj.bark();
7     cout<<dogObj.age<<" "<<dogObj.colour<<endl;
8     return 0;
9 }
```

BASE



OUTPUT:

- ✓ Eating
- ✓ Barking
- ✓ 8 Black

5.2.1 Mode of inheritance table

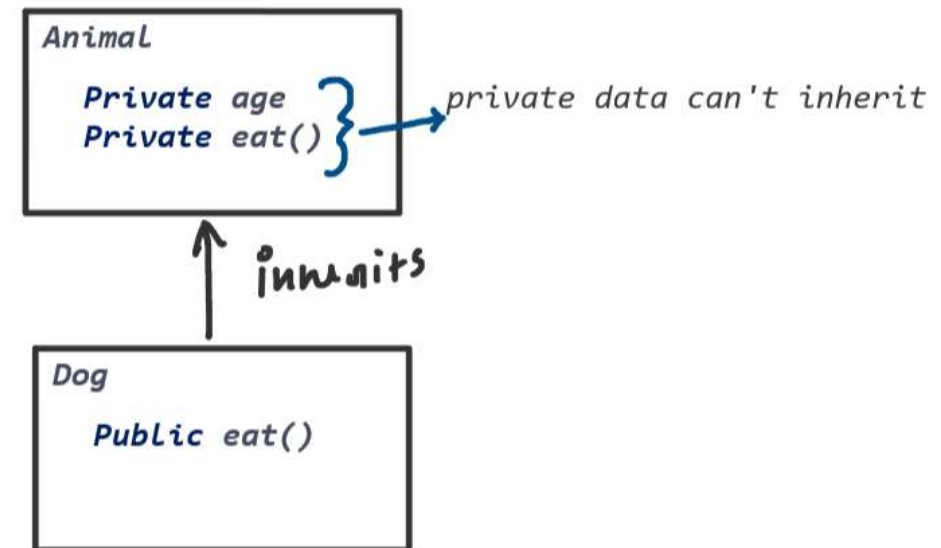
	Derived Class	Derived Class	Derived Class
Base Class	Private Mode	Protected Mode	Public Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

```

1 // Case 01: Private member inherits as private mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     private:
9         int age;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14 };
15
16 // Derived Class Dog
17 class Dog: private Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

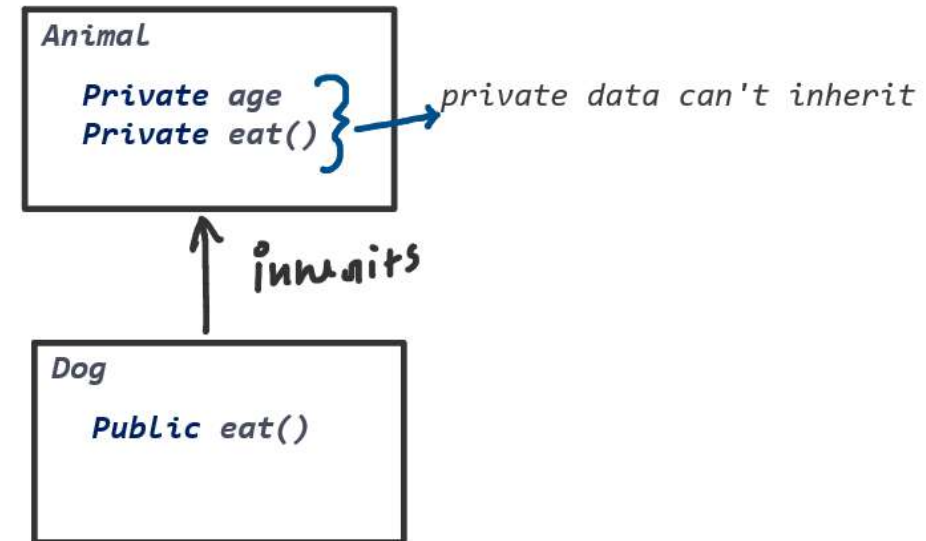



```

1 // Case 02: Private member inherits as protected mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     private:
9         int age;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14 };
15
16 // Derived Class Dog
17 class Dog: protected Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

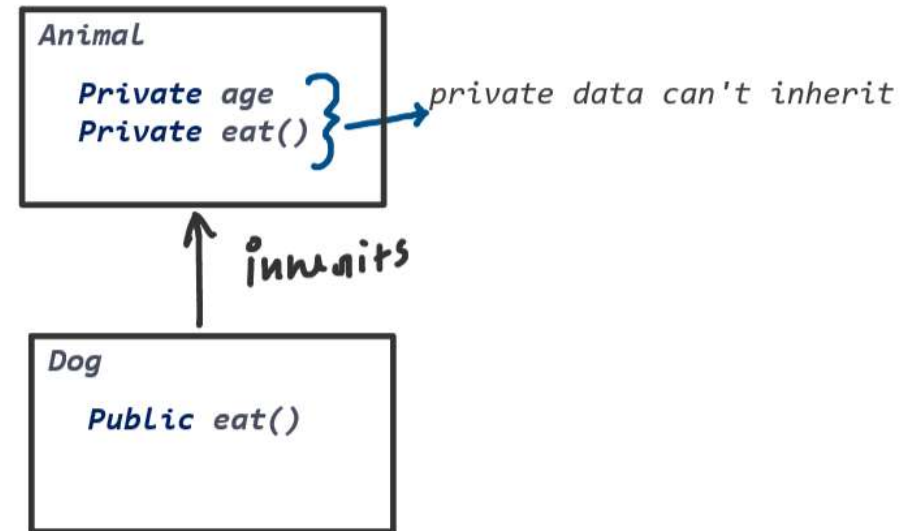


```

1 // Case 03: Private member inherits as public mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     private:
9         int age;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14 };
15
16 // Derived Class Dog
17 class Dog: public Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

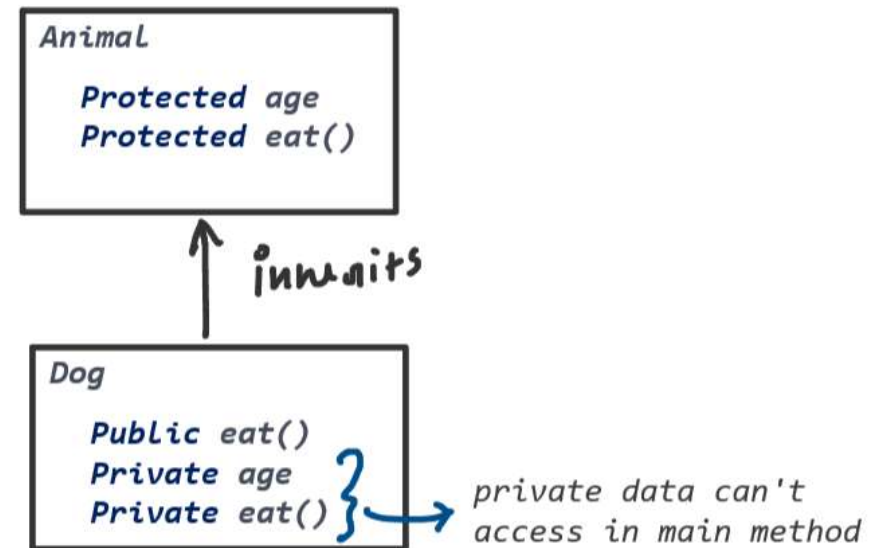


```

1 // Case 04: Protected member inherits as private mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     protected:
9         int age;
10
11     void eat(){
12         cout<<"Eating"<<endl;
13     }
14 };
15
16 // Derived Class Dog
17 class Dog: private Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

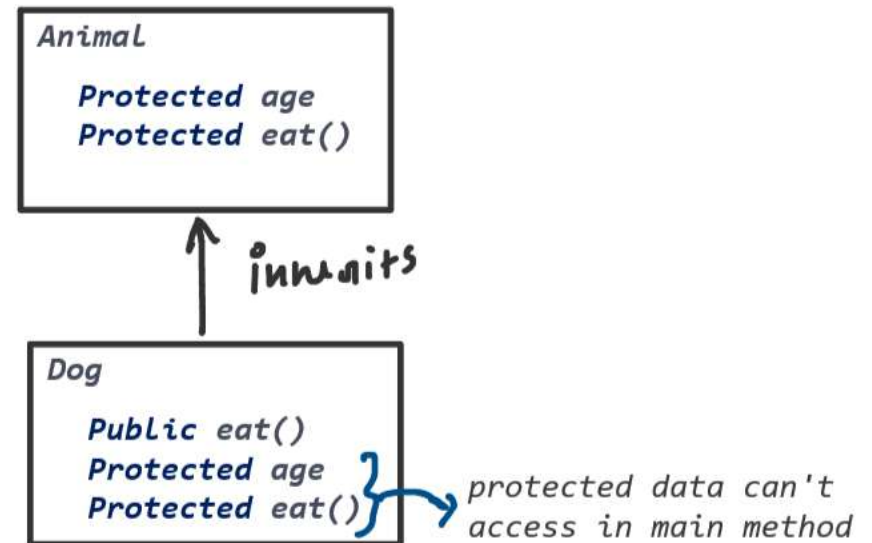


```

1 // Case 05: Protected member inherits as protected mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     protected:
9         int age;
10
11     void eat(){
12         cout<<"Eating"<<endl;
13     }
14 };
15
16 // Derived Class Dog
17 class Dog: protected Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

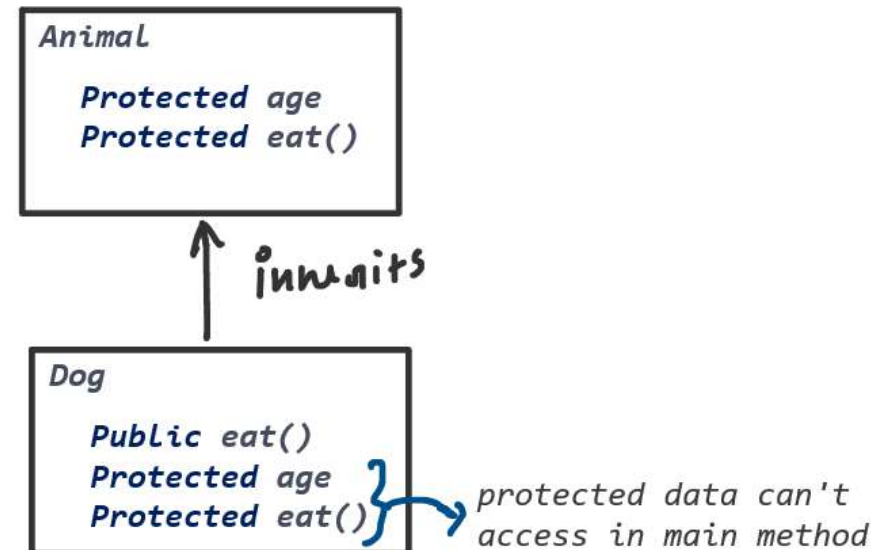


```

1 // Case 06: Protected member inherits as public mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     protected:
9         int age;
10
11     void eat(){
12         cout<<"Eating"<<endl;
13     }
14 };
15
16 // Derived Class Dog
17 class Dog: public Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

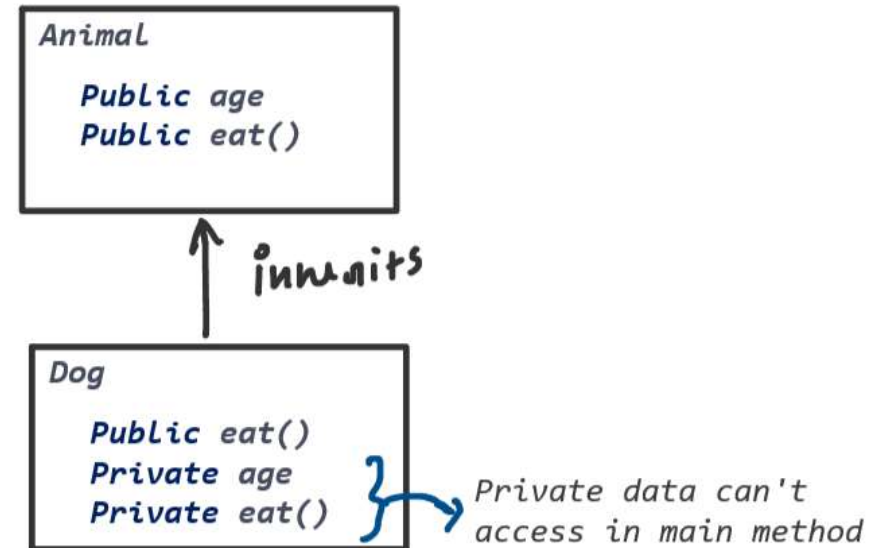


```

1 // Case 07: Public member inherits as private mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     public:
9         int age;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14 };
15
16 // Derived Class Dog
17 class Dog: private Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking

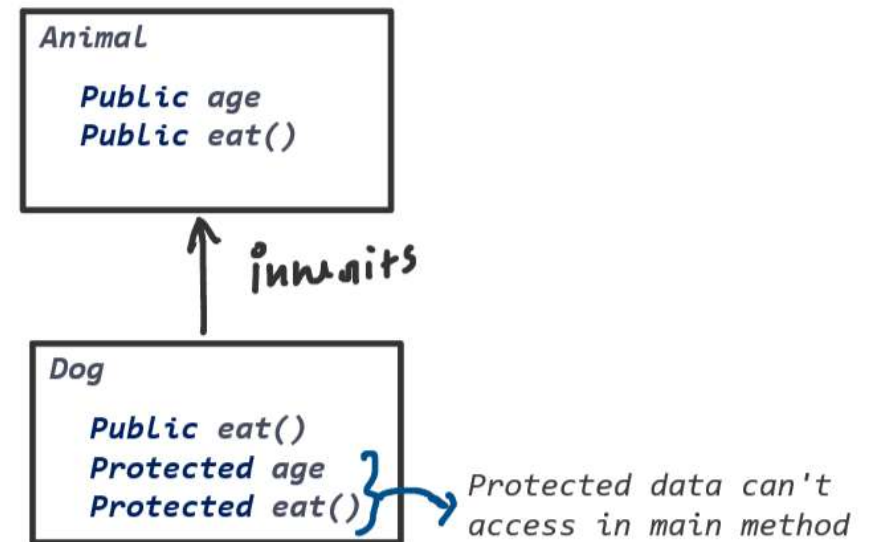


```

1 // Case 08: Public member inherits as protected mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     public:
9         int age;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14 };
15
16 // Derived Class Dog
17 class Dog: protected Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     return 0;
29 }

```

Output
Barking




```

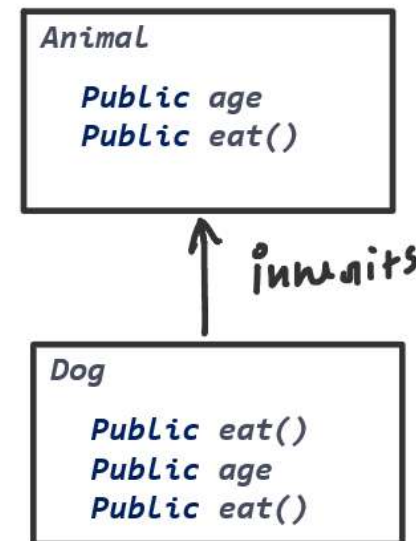
1 // Case 09: Public member inherits as public mode
2 #include<iostream>
3 using namespace std;
4
5 // Base Class Animal
6 class Animal
7 {
8     public:
9         int age;
10
11         void eat(){
12             cout<<"Eating"<<endl;
13         }
14 };
15
16 // Derived Class Dog
17 class Dog: public Animal
18 {
19     public:
20         void bark(){
21             cout<<"Barking"<<endl;
22         }
23 };
24
25 int main(){
26     Dog dogObj;
27     dogObj.bark();
28     dogObj.age = 8;
29     dogObj.eat();
30     return 0;
31 }

```

age [8]

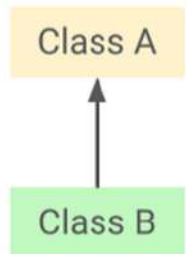
output

Barking
Eating

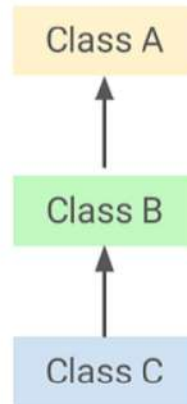


Public data can
access in main method

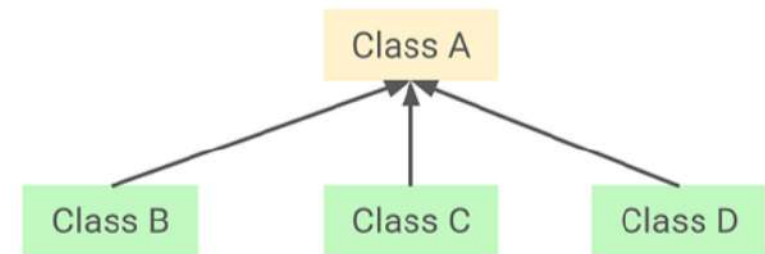
5.2.2 Type of inheritance



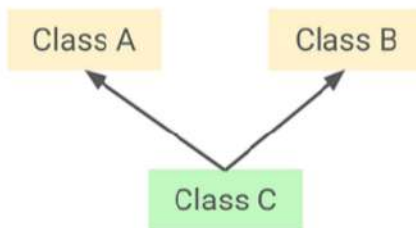
Type 01: Single



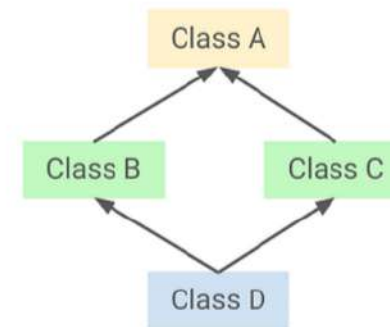
Type 02: Multilevel



Type 03: Hierarchical



Type 04: Multiple

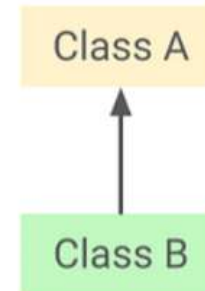


Type 05: Diamond Problem (Hybrid inheritance)

```

1 // 5.2.2.1 Single inheritance program
2 #include<iostream>
3 using namespace std;
4
5 // Base Class A
6 class A
7 {
8     public:
9         int id;
10
11         void funA(){
12             cout<<"FunA called"<<endl;
13         }
14 };
15
16 // Derived Class B
17 class B: public A
18 {
19     public:
20         void funB(){
21             cout<<"FunB called"<<endl;
22         }
23 };
24
25 int main(){
26     B Bobj;
27     Bobj.funA();
28     Bobj.id = 8;
29     Bobj.funB();
30     return 0;
31 }

```



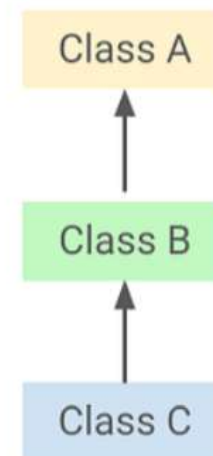
Type 01: Single

Output:

FunA called

FunB called

```
1 // 5.2.2.2 Multilevel inheritance program
2 #include<iostream>
3 using namespace std;
4
5 // Base Class A for B
6 class A
7 {
8     public:
9         int id;
10
11         void funA(){
12             cout<<"FunA called"<<endl;
13         }
14 };
15
16 // Derived Class B for A and Base class B for C
17 class B: public A
18 {
19     public:
20         void funB(){
21             cout<<"FunB called"<<endl;
22         }
23 };
24
25 // Derived Class C for B
26 class C: public B
27 {
28     public:
29         void funC(){
30             cout<<"FunC called"<<endl;
31         }
32 };
33
34 int main(){
35     B Bobj;
36     Bobj.funA();
37     Bobj.funB();
38     Bobj.id = 8;
39
40     C Cobj;
41     Cobj.funA();
42     Cobj.funB();
43     Cobj.funC();
44     Cobj.id = 10;
45     return 0;
46 }
```



Output:
FunA called
FunB called
FunA called
FunB called
FunC called

Type 02: Multilevel

```

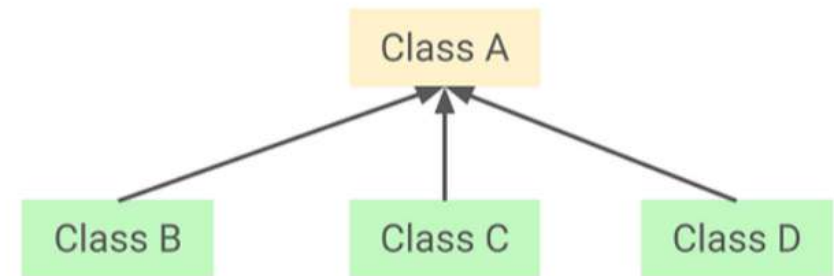
1 // 5.2.2.3 Hierarchical inheritance program
2 #include<iostream>
3 using namespace std;
4
5 // Base Class A for B, C, and D
6 class A
7 {
8     public:
9         int id;
10
11         void funA(){
12             cout<<"FunA called"<<endl;
13         }
14 };
15
16 // Derived Class B for A
17 class B: public A
18 {
19     public:
20         void funB(){
21             cout<<"FunB called"<<endl;
22         }
23 };
24
25 // Derived Class C for A
26 class C: public A
27 {
28     public:
29         void funC(){
30             cout<<"FunC called"<<endl;
31         }
32 };
33
34 // Derived Class D for A
35 class D: public A
36 {
37     public:
38         void funD(){
39             cout<<"FunD called"<<endl;
40         }
41 };

```

```

1
2 int main(){
3     B Bobj;
4     Bobj.funA();
5     Bobj.funB();
6     Bobj.id = 8;
7
8     C Cobj;
9     Cobj.funA();
10    Cobj.funC();
11    Cobj.id = 10;
12
13    D Dobj;
14    Dobj.funA();
15    Dobj.funD();
16    Dobj.id = 12;
17    return 0;
18 }

```



Type 03: Hierarchical

Output:

```

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

```

```

1 // 5.2.2.4 Multiple inheritance program
2 #include<iostream>
3 using namespace std;
4
5 // Base Class A as Teacher
6 class Teacher
7 {
8     public:
9         void teach(){
10             cout<<"Teaching"<<endl;
11         }
12 };
13
14 // Base Class B as Researcher
15 class Researcher
16 {
17     public:
18         void research(){
19             cout<<"Researching"<<endl;
20         }
21 };
22
23 // Derived Class C as Professor
24 class Professor: public Teacher, public Researcher
25 {
26     public:
27         void bore(){
28             cout<<"Boring"<<endl;
29         }
30 };

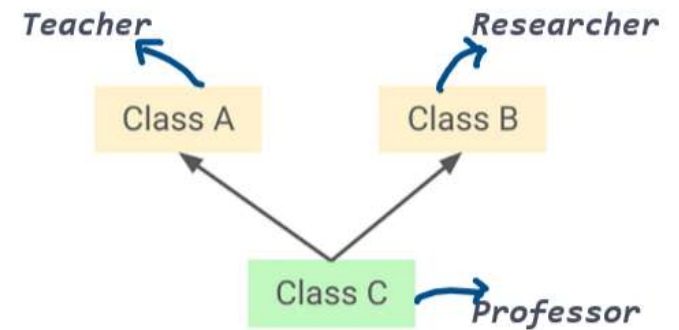
```

```

1 int main(){
2     Professor PObj;
3     PObj.bore();
4     PObj.teach();
5     PObj.research();
6     return 0;
7 }

```

Output:
 Boring
 Teaching
 Researching

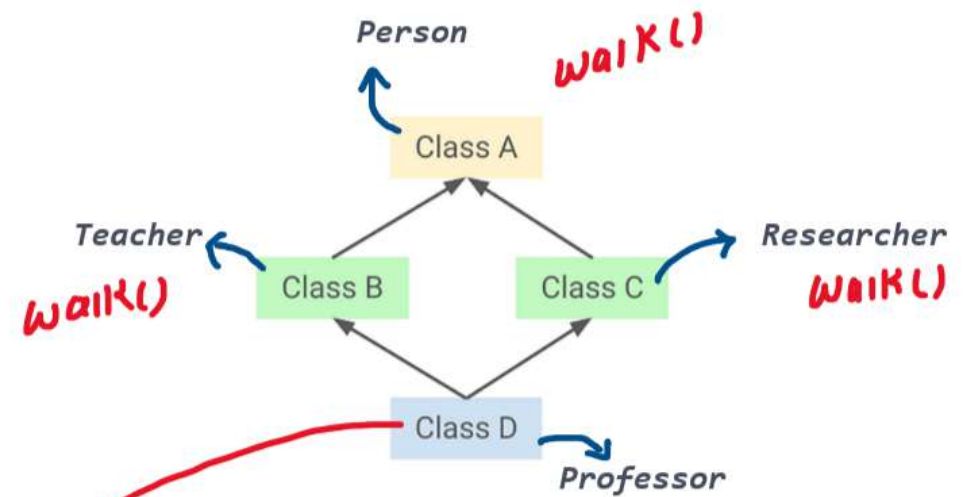


Type 04: Multiple

```

1 // 5.2.2.5 Diamond Problem (Hybrid inheritance)
2 #include<iostream>
3 using namespace std;
4
5 // Base Class A as Person for Teacher and Researcher
6 class Person
7 {
8     public:
9     void walk(){
10         cout<<"Walking"<<endl;
11     }
12 };
13
14 // Derived class B as Teacher for Person and Base Class Teacher for Professor
15 class Teacher: public Person
16 {
17     public:
18     void teach(){
19         cout<<"Teaching"<<endl;
20     }
21 };
22
23 // Derived class C as Researcher for Person and Base Class Researcher for Professor
24 class Researcher: public Person
25 {
26     public:
27     void research(){
28         cout<<"Researching"<<endl;
29     }
30 };
31
32 // Derived Class D as Professor for Teacher and Researcher
33 class Professor: public Teacher, public Researcher
34 {
35     public:
36     void bore(){
37         cout<<"Boring"<<endl;
38     }
39 };
40
41 int main(){
42     Professor PObj;
43     PObj.walk(); // error: request for member 'walk' is ambiguous
44     return 0;
45 }

```



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


```

1 // 5.2.2.5.1 Diamond Problem (Hybrid inheritance) with Scope resolution
2 #include<iostream>
3 using namespace std;
4
5 // Base Class A as Person for Teacher and Researcher
6 class Person
7 {
8     public:
9     void walk(){
10         cout<<"Walking"<<endl;
11     }
12 };
13
14 // Derived class B as Teacher for Person and Base Class Teacher for Professor
15 class Teacher: public Person
16 {
17     public:
18     void teach(){
19         cout<<"Teaching"<<endl;
20     }
21 };
22
23 // Derived class C as Researcher for Person and Base Class Researcher for Professor
24 class Researcher: public Person
25 {
26     public:
27     void research(){
28         cout<<"Researching"<<endl;
29     }
30 };

```

Ambiguous Problem Error Solution 01

```

1 // Derived Class D as Professor for Teacher and Researcher
2 class Professor: public Teacher, public Researcher
3 {
4     public:
5     void bore(){
6         cout<<"Boring"<<endl;
7     }
8 };
9
10 int main(){
11     Professor PObj;
12     // Diamond Problem
13     // Solution 1: Scope Resolution
14     PObj.Teacher::walk();
15     PObj.Researcher::walk();
16     return 0;
17 }

```

Output:
Walking
Walking

Not Good Sol.ⁿ

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

```

1 // 5.2.2.5.2 Diamond Problem Solution Using Virtual
2 #include<iostream>
3 using namespace std;
4
5 class Person
6 {
7     public:
8     void walk(){
9         cout<<"Walking"<<endl;
10    }
11 };
12
13 class Teacher: virtual public Person
14 {
15     public:
16     void teach(){
17         cout<<"Teaching"<<endl;
18     }
19 };
20
21 class Researcher: virtual public Person
22 {
23     public:
24     void research(){
25         cout<<"Researching"<<endl;
26     }
27 };
28
29 class Professor: public Teacher, public Researcher
30 {
31     public:
32     void bore(){
33         cout<<"Boring"<<endl;
34     }
35 };

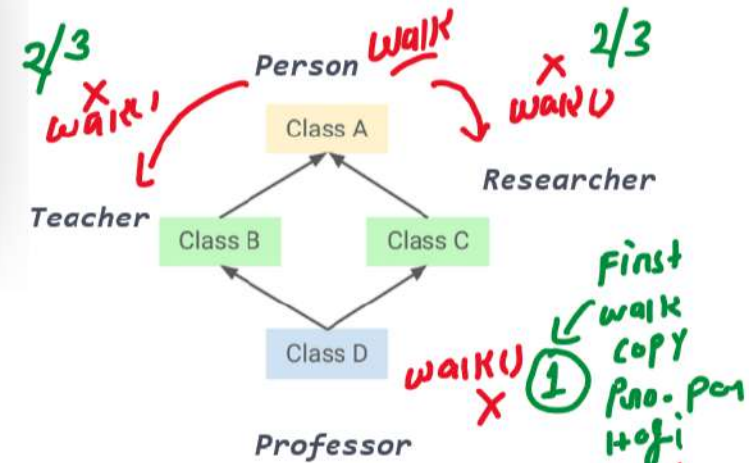
```

```

1 int main(){
2     Professor PObj;
3     // Diamond Problem
4     // Solution 2: using virtual
5     PObj.walk();
6     return 0;
7 }

```

Ambiguous Problem Error **Solution 02**



Output:
Walking

Compiler Time par walk() ki copy nahi milti hai
Teacher, Researcher, Prof. - In sabko Runtime
par copy assign ki jati hai.

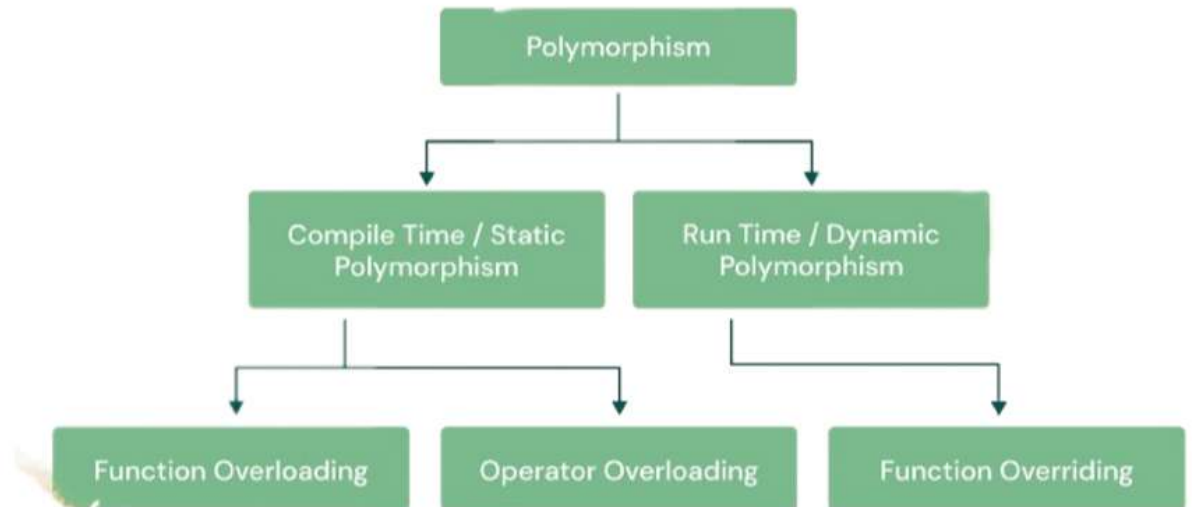


5.3 Polymorphism

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:

```
1 // (I.) Function Overloading Program
2 #include <iostream>
3 using namespace std;
4
5 // First fn
6 int add(int a, int b)
7 {
8     return a + b;
9 }
10
11 // Second fn: different types of parameters from first
12 double add(double a, double b)
13 {
14     return a + b;
15 }
16
17 // Third fn: different number of parameters from first
18 int add(int a, int b, int c)
19 {
20     return a + b + c;
21 }
22
23 int main(){
24     cout<< add(5, 10) << endl;
25     cout<< add(5.5, 10.5) << endl;
26     cout<< add(5,10, 15) << endl;
27     return 0;
28 }
```

(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 ⊕

```
1 // (II.) Operator Overloading Program
2 #include <iostream>
3 using namespace std;
4
5 class Vector
6 {
7     private:
8         int x, y;
9
10    public:
11        // Init list CTOR
12        Vector(int x, int y): x(x), y(y){}
13
14        // Simple member fn
15        void display(){
16            cout<<"x: "<<x<<" y: "<<y<<endl;
17        }
18
19        // Operator overloading
20        void operator+(const Vector &src)
21        {
22            // this would point to v1
23            // src would point to v2
24            this->x += src.x;
25            this->y += src.y;
26        }
27 };
```

```
1 int main(){
2     Vector v1(2, 3);
3     Vector v2(4, 5);
4
5     cout<<"Before operator overloading"<<endl;
6     v1.display();
7     v2.display();
8
9     cout<<"After operator overloading"<<endl;
10    v1 + v2;
11    // Additional ans (v1+v2) should be stored in v1
12    v1.display();
13    v2.display();
14
15    return 0;
16 }
```

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

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

$$V_1 + V_2 \Rightarrow \left. \begin{matrix} \text{src} = V_2 \\ \text{dst} = V_1 \end{matrix} \right\} \Rightarrow \text{means } \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \end{bmatrix} \\ V_1 = V_1 + V_2$$

Ex

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

Destination
→ V_1 \oplus $V_2 \rightarrow$ Source

Addition operation (+)

Void **Operator** + (const Vector **&src**)

{

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

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

{
this → x = this → x + src · x ;
this → y = this → y + src · y ;

}

Destination
→ V_2

$-$ $V_1 \rightarrow$ source

Subtract operation (-)

Void **operator** - (const Vector &src)

{

this $\xrightarrow{\text{points}}$ V_2
src $\xrightarrow{\text{points}}$ V_1

{
this $\rightarrow x = \text{this} \rightarrow x - \text{src} \cdot x$;
this $\rightarrow y = \text{this} \rightarrow y - \text{src} \cdot y$;
}

}

Ex:2 ☹️

```
1 // (II.) Operator Overloading Program
2 #include <iostream>
3 using namespace std;
4
5 class Vector
6 {
7     private:
8         int x, y;
9
10    public:
11        // Init list CTOR
12        Vector(int x, int y): x(x), y(y){}
13
14        // Simple member fn
15        void display(){
16            cout<<"x: "<<x<<" y: "<<y<<endl;
17        }
18
19        // Operator overloading
20        void operator-(const Vector &src)
21        {
22            // this would point to v2
23            // src would point to v1
24            this->x -= src.x;
25            this->y -= src.y;
26        }
27 };
```

```
1 int main(){
2     Vector v1(2, 3);
3     Vector v2(4, 5);
4
5     cout<<"Before operator overloading"<<endl;
6     v1.display();
7     v2.display();
8
9     cout<<"After operator overloading"<<endl;
10    v2 - v1;
11    // Subtraction ans (v2-v1) should be stored in v2
12    v1.display();
13    v2.display();
14
15    return 0;
16 }
```

$$\begin{matrix} \begin{bmatrix} 4 \\ 5 \end{bmatrix} \\ v_2 \end{matrix} - \begin{matrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} \\ v_1 \end{matrix} = \begin{matrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} \\ v_2 = v_2 - v_1 \end{matrix}$$

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. `sizeof`,
2. `.` (member selection)
3. `?:` (conditional)
4. `::` (scope resolution)
5. `new`
6. `delete`