Month – 02 , class – 02

C++ oop class – 07

Inheritance and it’s types

---

1. Multi-level Inheritance

in multi-level inheritance, one class inherits from another derived class.

the relation continues in a chain like Grandpa → Dad → Son.

```
class Grandpa {
    int age;
public:
    void setter(int a) {
        age = a;
        cout<<"Grandpa setter"<<endl;
    }
    void getter() {
        cout<<"It's Grandpa class"<<endl;
        cout<<"Age : "<<age<<endl;
    }
};

class Dad : public Grandpa {
    int net_worth;
public:
    void setter(int n) {
        net_worth = n;
        cout<<"Dad setter"<<endl;
    }
    void getter() {
        cout<<"It's Dad class"<<endl;
```

```
        cout<<"Net_worth : "<<net_worth<<endl;
    }
};

class Son : public Dad {
public:
    void getter() {
        cout<<"It's Son class"<<endl;
    }
};
```

here, each class defines its own setter and getter.
Son inherits both Grandpa and Dad but uses its own version of getter.
the last class in the chain can access everything public or protected from the previous ones.

2. Constructor and Destructor Order in Multi-level Inheritance

in a chain like Grandpa → Dad → Son, constructors are called from base to derived, and destructors in the reverse order.

```
class Grandpa {
public:
    Grandpa() { cout<<"Grandpa constructor"<<endl; }
    ~Grandpa() { cout<<"Grandpa destructor"<<endl; }
};

class Dad : public Grandpa {
public:
    Dad() { cout<<"Dad constructor"<<endl; }
    ~Dad() { cout<<"Dad destructor"<<endl; }
};

class Son : public Dad {
public:
    Son() { cout<<"Son constructor"<<endl; }
```

```
    ~Son() { cout<<"Son destructor"<<endl; }
};
```

output shows:
Grandpa constructor
Dad constructor
Son constructor
Son destructor
Dad destructor
Grandpa destructor

this proves constructors run from base to derived and destructors in reverse.

Grandpa → Dad → Son for constructors
Son ← Dad ← Grandpa for destructors.

3. Hierarchical Inheritance

in hierarchical inheritance, one base class is inherited by more than one derived class.
the base constructor runs first, then the constructor of the derived object being created.
each derived class works independently.

```
class A {
public:
    A() { cout<<"A cons"<<endl; }
    ~A() { cout<<"A des"<<endl; }
};

class B : public A {
public:
    B() { cout<<"B cons"<<endl; }
    ~B() { cout<<"B des"<<endl; }
};
```

```
class C : public A {
public:
    C() { cout<<"C cons"<<endl; }
    ~C() { cout<<"C des"<<endl; }
};
```

creating an object of B prints

A cons

B cons

B des

A des

creating an object of C prints

A cons

C cons

C des

A des

destructor always gets called in the reverse order of constructor.

4. Multiple Inheritance

multiple inheritance means one class inherits from more than one base class.
constructor order follows the order of base classes written in the inheritance list, and destructors reverse that order.

```
class A {
public:
    A() { cout<<"A cons"<<endl; }
    ~A() { cout<<"A des"<<endl; }
};

class B {
public:
```

```
    B() { cout<<"B cons"<<endl; }
    ~B() { cout<<"B des"<<endl; }
};

class C : public B, public A {
public:
    C() { cout<<"C cons"<<endl; }
    ~C() { cout<<"C des"<<endl; }
};
```

output sequence for constructors:

B cons

A cons

C cons

destructors in reverse:

C des

A des

B des

5. Hybrid Inheritance

hybrid inheritance combines more than one type of inheritance.

in this code, hierarchical and multiple inheritance exist together.

A is a base class for both B and C, and D inherits from both B and C.

```
class A {
public:
    A() { cout<<"A cons"<<endl; }
    ~A() { cout<<"A des"<<endl; }
};
```

```
class B : public A {
public:
    B() { cout<<"B cons"<<endl; }
    ~B() { cout<<"B des"<<endl; }
};

class C : public A {
public:
    C() { cout<<"C cons"<<endl; }
    ~C() { cout<<"C des"<<endl; }
};

class D : public B, public C {
public:
    D() { cout<<"D cons"<<endl; }
    ~D() { cout<<"D des"<<endl; }
};
```

constructor order here is

A cons

B cons

A cons

C cons

D cons

and destructor order reverses that sequence.

A → B , C (hierarchical)

D ← B , C (multiple)

this combination forms hybrid inheritance.

6. Runtime and Compile-time Polymorphism

compile-time polymorphism includes function overloading and operator overloading.
runtime polymorphism is achieved through **function overriding** with the help of virtual functions.

```
class A {
public:
    virtual void display() {
        cout<<"A function"<<endl;
    }
};

class B : public A {
public:
    void display() override {
        cout<<"B function"<<endl;
    }
```