CO-2

12/08/24

OOP Features

① **Encapsulation** - Wrapping up of data & methods into a single unit is called encap.

Class – Keyword, Point – Identifier.

```
Class Point
{  int x;
   int y;
   public:
   void set-x (int a)
   {  x = a
   }
   void set-y (int b)
   {  y = b
   }
};
```

3 types of access specifiers : ① private ② protected ③ public

[By default it is PRIVATE]

[If a func^n is defined within a class it is INLINE func.]

```
void print()
{ cout << x << endl << y;
}
```

```
int main () {
    Point ob;
    ob. set-x (10);
    ob. set-y (20);
    ob. print();
}
```

[x, y are private
~~cout << ob.x << ob.y;~~]

```
int main()
{ Point ob[10];
  for (int i=0; i<10; i++) {
      ob[i]. set-x (i);
      ob[i]. set-y (i*i);
  }

  for (int i=0; i<10; i++) {
      ob[i]. print();
  }
}
```

```
class Point
{ int x, y;
  public:
  void set-val (int x, int y)
  {  this→x = x;
     this→y = y;
  }

  void display ()
  {  cout << (*this).x;
     cout << (*this).y;
  }
};
```

["this→" ≡ "*this."]

```
int main()
{ Point ob;
  ob. set-val (5,15);
  ob. display();
}
```

Even if we don't write any constructor, a default constructor is generated.

CONSTRUCTOR → It has name, same as class' name.
→ It never has return type.

```
class Point
{ int x,y;
  public:
    Point()  ──→ No parameters ──→ Default Constructor
    { x=10;        No Argum. or const.
      y=10;
    }
    Point(int x)
    { this->x = x;
      this->y = x*x;
    }
    Point(int x, int y)
    { this->x = x;
      this->y = y;
    }
```

Constructor Overloading

```
int main() {
  Point ob1;
  ob1.disp();
  Point ob2(5);
  ob2.disp();
  Point ob3(10,20);
  ob3.disp();
```

If no const. → compiler supplies 1 default const.

If we supply atleast 1 parameterised const. → No default const is provided.

Supply default values from RIGHT to LEFT.

```
class Point
{ int x;
  const int y=5;        One way ⇒ const int y=5
  public:
    void set()
    { x=10;
      y=5;   → It is constant & cannot be initialised & within member func".
    }
    void show() const   → READ ONLY METHOD
    { cout<<"x="<<x;      It can only read & can't change values.
      cout<<"y="<<y;
      x=10;
```

READ ONLY member method } can't modify.

```
int main()
{ Point ob;
  ob.set();
  ob.show();
}
```

Const. method can be called in normal way

```
class Point
{ int x;
  const int y;
  public:
    Point(int x): y(5)   → 2nd Way.
    { this->x = x;
    }
```

→ Not possible.

```
    void set(int x, int y)
    { this->x = x;
      this->y = y;
    }
```

```
class Point
{ mutable int x;
  const int y = 10;

public:
  void incr() const
  { x++;   ---> Possible as x is mutable
    y++;   ---> const.
  }

int main()
{ const Point ob;   ---> const. class object.
                    ---> A const. object can only call const. member methods.
  ob.set(2,5);
  ob.incr();
```

A const. member method can modify only mutable variables

14/04/24

```
class AB
{ int a,b;
  public:
    void setA(int x)
    { a = x;
      return *this;
    }
    void setb(int y)
    { b = y;
      return *this;
    }
    void disp()
    { cout << "a = " << a;
      cout << "a. " << a;
```

Cascaded func^n call

```
int main() {
{ AB ob;
  ob.set_a(5).set_b(7)
   .disp();
  ob.set_a(5), set_b(10).
   .disp();
```

Now it runs.

If we return this & not *this, then

```
AB* set_a(int x) {
  a = x
  return this;
}
```

⇒ ob.set_a(5) ⟶ set_b(10) ⟶ disp();

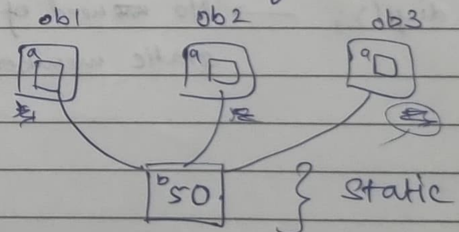
```
class AB
{ int a;
  static int b;
  public:
    AB(int x)
    { a = x;
      b = 100   → also possible.
    }           but have to write int AB::b;
    void disp()
    { cout << "a = " << a;
      cout << "b = " << b;
    }
};
int AB::b = 100;
```

if not, LINKER ERROR

Memory of static variable is

```
int main()
{ AB ob(10);
  ob.disp();
```

⇒ if we write static as public, we can access the var. by, cout << AB::b << endl;



} Static variable is shared.

14/8/24

class has b++; (b:50).

AB ob1, ob2, ob3;

53

Static Member Method

```
class AB
{   int a;
    static int b;
    public:
        AB (int x)
        {  a = x;
           b++;
        }
        static void s_disp()
        {  cout << "a = " << a;
           cout << "b = " << b;
        }
};
int AB :: b = 100;

int main ()
{  AB ob (10);
   AB :: s_disp();
```

Static member methods are prop. of class

ERROR, we can't access non-static member variables from static member methods

Existence of non-static variable depends on existence of obj.

→ No need of object to call static member func^{n}.

AB :: f2();

```
class AB
{ static int a;
  public:
    AB (int x = 0)
    {  a = x;
    }
    static void f1()
    {  cout << a << endl;
       f2();
    }
    static void f2()
    {  cout << a+1 << endl;
    }
};
int AB :: a;
```

A static member method cannot access this operator (refers to current obj. of class)

Static member func^{n} can access static variable only.

⇒ To call non-static func^{n} in static func^{n}.

```
static void f1() {
    AB ob;
    ob.f2();
}
```

→ non-static func^{n}.

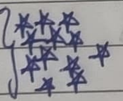
T.O.

```
class AB
{  static int s;
   int a;
   public :
      AB()
      {  s++;
         a = s;
      }
      void disp()
      {  cout << "a = " << a;
         cout << "b = " << b;
      }
};
int AB :: s = 0;

int main() {
   AB obj[5];
   for (int i = 0; i < 5; i++) {
      cout << obj[i].disp();
   }
```

→ On declaring immediately memory is allocated.

Output : a = 1, 2, 3, 4, 5
s = 5, 5, 5, 5, 5



```
class AB
{  int a, b;
   public :
      void set_a(int);
      void set_b(int);
      void disp();
};

void AB :: set_a (int x)
{  a = x;
}
void AB :: set_b (int y)
{  b = y;
}
void AB :: disp()
{
}
```

function Headers.

Only for public func".

```
class AB
{  int a, b;

   public :
   AB (AB &ob)
   {  a = ob.a;
      b = ob.b;
   }
```

→ Pass by reference.

Copy Constructor.

~AB() → Destructor.

AB ob2 = ob1

AB ob2(ob1);

21/8/24

```
class A
{ int a;
  public :
    void disp () {
      cout << "a = " << a ;
    }
    int get_a() {
      return a;
    }
};

int main() {
    A oba (5);
    B obb (7);
    cout << oba.get_a() + obb.get_b() << endl;
}      OR
    obb. add (&oba);
```

Swapping

```
int temp = oba.get_a();
oba.set_a(obb.get_b());
obb.set_b( temp);
        or
obb. swap (&oba)
```

```
class B
{ int b;
  public :
    void disp() {
      cout << "b = " << b;
    }
    int get_b() {
      return b;
    }
    void add (A obj) {
      int sum;
      sum = b + obj.get_a();
      cout << sum << endl;
    }

void swap (A &ob)
{ int t = ob.get_a();
  ob.set_a(&b);
  b = t ;
}
```

Address

NOT a MEMBER FUNCTION

Never use Scope res. operator to define the func".

→ It can access private member variables directly. in which classes it is defined as friend.

21/8/24

Friend function (used bcoz earlier is getting hectic);

Acts as a bridge b/w 2 or more unrelated classes.

```
class A {
  int a;
  public :
  friend void swap (A&, B&) ;
};

class B {
  int b;
  public :
  friend void swap (A&, B&) ;
};
```

```
void swap (A &oba, B &obb) {
  int t = oba.a ;
  oba.a = obb.b ;
  obb.b = oba.a ;
}
```

(should be called just like a GLOBAL function)

⇒ It ~~happens~~ hampers Data Abstraction.

```
class Dist
{ int feet;
  int inches;
  public:
    Dist ( int f=0, int i=0) {
      feet = f ;  inches = i ;
    }
    Dist add (Dist &ob)
    { Dist t ;
      t.feet = feet + ob.feet;
      t.inches = inches + ob.inches ;

      t.feet += t.inches / 12 ;
      t.inches %= 12 ;
```

→ No default. const.

__/__/____

*non-virtual

non-virtual

B( )

C1(int) : B()

C2( ) : B()

B(int x)

GC( ) : c1( ), c2( )

In non-virtual, the parameterised is called.

In virtual, the default const. is called always, if not existing then ERROR.

In virtual, in GC : c1(), c2(), B()
then it calls parameterised only.

class B

class C : public B

class GC : public C

virtual void show()

show()

show()

It is propagated

B *p = new GC(8) ⟶ GC→show();

p→show()

Mc and the [redacted]!

__/__/____

CO-4 Operator Overloading.

* Operators which can be OVERLOADED.

BINARY ⟶

1. Arithmetic: +, -, *, /, %, +=, -=, *=, /= — USING MEMBER AND FRIEND FUNCN.
2. Relational: <, <=, >, >=, !=, ==
3. logical: &&, ||, !
4. Bitwise: &, |, ^, ~, <<, >>
5. Mixed mode arith.: ob1 = ob2 + 5.
6. →
7. ( )
8. [ ]
9. cin>> cout<<
10. conversion op.: BDT & UDT
11. new, delete.

UNARY ⟶

12. pre/post inc.

* Operators which can't be Overloaded.

1. sizeof()
2. ::
3. . (dereferencing) (dot)
4. * (pointer deref.)
5. ?: [conditional op.]

```
class AB
{ int a;
  public:
    AB (int x = 0)
    { a = x;
    }
    AB operator+ (AB &ob)
    { AB t;
      t.a = a + ob.a;
      return t;
    }
```

```
int main()
{ AB a1(5), a2(6), a3;
  a3 = a1 + a2;
  a3.disp();
```

a3 = a1.operator+(a2);

if

In virtual public inheritance, the base class has default and parameterised, the default of base class is ~~if~~ ALWAYS CALLED., even if you mention (: B(x))

If there is no default cons. in base class, the prog. will result in an ERROR.

Soln: G(int z) : C2(z), C1(z+1), B(z)

We call Base ~~cls~~ explicitly.

04/09/24

```
class B                          class C: public B
{ int x;                         { int y;
  public:                          public:
    B (int a)                      C (int b) : B(b+1)   // Cuz base class const. is parameterised.
    { x = a;                       { y = b;
    }                              }
    virtual void show() = 0;       void show()
};                                 { B::show();
                                   { cout << "y=" << y;
void B:: show() {                  }
    cout <<  .. }; };              ~C()
                                   { cout << "C dest";
int main() {                       }
  B *p = new C(5);   } Error if no virtual
  p -> show();       }
} delete p. ——→ (Base class constructor is called, not the child class)
```

virtual ~B()
{ cout<<
}

★ A class containing Pure Virtual Func / Pure Virt. Destruc. is called an ~~Abstract~~ class (which means it can't be instantiated).

If base class dest. VIRTUAL, 1st child destr. is called, then the base class dest. is called.

using scope resol. oper.

A pure virtual dest. must be defined outside the base class, otherwise LINKER ERROR is shown.

```
class B                          class C : public B
{ int x;                         { int p;
  protected:                       public:
    int y;                         C (int m, int n) : B(m+n)
  public:                          { p = m; y = n;
    B (int a)                      }
    { x = a;                       void show()
    }                              { cout << "p=" << p;
  ~B() void show()                   cout << "y=" << y;
    { cout << "x = " << x;         }
    }
};                                 class | C ob (5,6)
                                   ob. show()
int main() {                       Calls this.
  B * P[4];
  for (int i=0; i<4; i++) {
      P[i] = new C(i, i+1)
  }
  for (int i=0; i<4; i++) {
      P[i] -> show();  ——→ calls show() of B
  }
```

To avoid Diamond Problem in Hybrid Inheritance, virtual public ~~static~~ mode is given to child. & called only once in shared memory.

30/8/24

```
class B                          class C: public B
{ public:                        { public:
    B()          (1)                 C()      (2)
    { cout << "B cont";              { cout <<
    }                                }
virtual ~B()     (4)                 ~C()     (3)
    { cout << "B dest"               {
    }                                }
}
```

```
int main()
{   B *ptr = new C();   } non virtual ~B
    delete ptr;          (1) → (2) → (4)
}                        virtual → 1 → 2 → 3 → 4
```

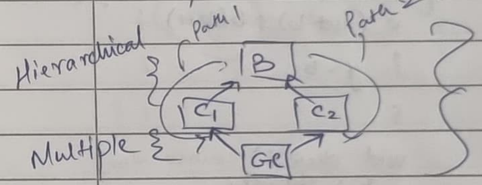
if destructor is pure virtual, its body has to be defined outside class using scope resolut°.
Otherwise, LINKER ERROR.

02/9/2024 MULTI-LEVEL INHERITANCE

```
class B                          class C: public B
{ int a;                         { int p;
  protected:                       protected:
    int b;                           int q;
  public:                        public:
    B(int x) {                       C(int y) : B(y)
    { a=x;                           { p=y; b=y+1;
    }                                }
};                               };
```

```
class GC : public C
{ int x;
  public:                                  Explicit call
    GC(int z) : C(z)
    { x=z;
      q = z+1;   b=z+2;
    }
};

int main()                                 → calls show() of B.
{  B *p = new GC(5);
   p → show();
}
```



HYBRID INHERITANCE

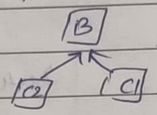
```
class B                  class C1: public B       class GC: public C1,
{                        {                                public C2
  protected:             }                        {
    int x;                                        public:
}                                                   GC()
                                                    { x = 10;
                                                    }
                                                 }
class C2: public B
{
};
```

If inheritance is virtual public, then one copy is made which is shared among others.

Non default constructor. C() : B1(5), B2(6)

Order of Inheritance is followed (in heading of class C).

Hierarchical Inheritance : Two Child, One Base




```
class B
{ int a;
  public:
   B(int x)
   { a=x;
   }
   void disp()
   { cout<< "a= "<<a;
   }

class C1 : public B
{ int b;
  public:
   C1() : B(10)
   { b=5;
   }
   void disp()
   { cout<< "b= "<<b;
   }

class C2 : public B
{ int c;
  public:
   C2(int x) : B(2*x*x)
   { c=x;
   }
   void disp()
   { cout<< "c= "<<c;
   }
}

int main() {
   C1 ob1;
   C2 ob2(100);
   ob1.disp();
   ob2.disp();
```

```
class B
{ public:
  virtual void disp()
   { cout << "B disp";
   }
}

class C : public B
{ public:
   void disp()
   { cout << "c disp";
   }

int main() {
   B *ptr;
   ptr = new C();
   ptr -> disp();
}
```

Run Time Polymorphism is not achieved.

After making base fun" VIRTUAL, then child class fun" is invoked.

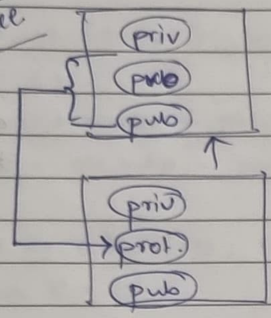
```
class B
{ public:
   virtual void fun() = 0;
```

ABSTRACT CLASS → which cant be instantiated.

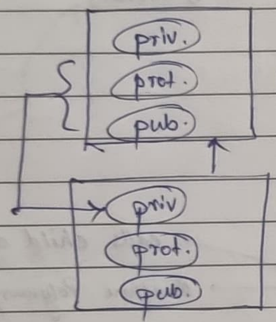
→ PURE VIRTUAL FUNC"

A pure virtual func" must be defined in CHILD class or else child class will also become ABSTRACT.

Protected Mode of Inheritance



Private Mode of Inheritance.




```
class B
{ int a;
  protected :
    int b;
  public : int p
    B () {
      b = p;
    }
  void disp () {
    cout << "a: " << a;
  }
```

```
class C : public B
{ int x;
  protected :
    int y;
  public :
    C (int a) : B(a)
    { x = a;
      y = a+1;
      b = a+2;
    }
  void disp() {
  cout << "x = "
       << "y = "
```

```
int main() {
```

Destructor is called in reverse order of constructors.

MULTIPLE INHERITANCE

```
class B1
{ public:
    B1()
    { cout<<"B1 cout";
    }
    void show()
    { cout<<"Hi!";
    }
```

```
class B2
{ public:
    B2()
    { cout <<"B2 cout";
    }
    void show ()
    { cout << "Hello!";
    }
```

```
class C : public B1, public B2
{ public:
    C()
    { cout<< "C cout";
    }
    void show() {
      B1:: show(); B2:: show();
      cout <<
    }
```

```
int main () {
  C ob;
  ob.show();
  ob. B1:: show();
  ob. show();
```

Ambiguous situation.

__/__/____

```
friend Dist add (Dist ob1, Dist ob2)
{   Dist t;
    t.feet = ob1.feet + ob2.feet;
    t.inches = ob1.inches + ob2.inches;

    t.feet += t.inches/12;
    t.inches %= 12;
}
```

```
class A {
    int a;
    public:
```

```
class B {
```

★ friend class B;
}

★ B can access private member variables through function.

★ friend func^{n} cannot access this→ operator as it is a non-member function.

30/8/24

INHERITANCE

public mode of inheritance

```
class B
{ int b;
  public:
    void set_b (int x) {
        b = x; }
    void show() {
        cout << "b = " << b << endl;
    }
```

```
class C : public B
{ int c;
  public:
    void set_c (int x) {
        c = x; }
    void show() {
        cout << "c = " << c;
    }
```

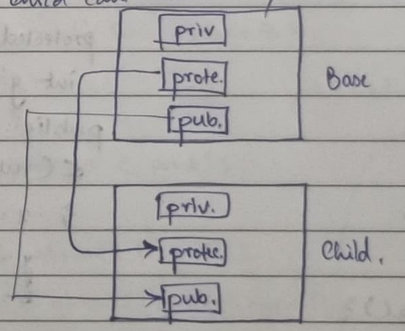
B::show();

```
int main() {
    C ob;
    ob.set_c (8);
    ob.set_b (10);
    ob.show();
```

→ calls child class fun^{n}. func^{n}

→ Runtime Polymorphism as signatures are same.

★ Even the child can't access the private variables of base class.



Protected can be accessed from child classes but not from main().