# Object Oriented Programming in C++

4 pillars →
- Abstraction → Hiding internal details
- Encapsulation → to bundle data and functions together
- Polymorphism → same name multiple functionalities
- Inheritance → Reusing code

Class → A data type with functions

Object → Variable of a class

```
class Complex
{ private:
    int real;
    int imag;
  public:
    void print()
    { cout<<real<<"+i"<<imag<<endl;
    }
    Complex(int r, int i)   ← Constructor
    {
        real=r;
        imag=i;
    }
};
int main()
{
    Complex C1(10,15);
    C1.print();
    return 0;
}
```

# Constructors and Destructors

```
class Point
{
  private:
    int x, y;
  public:
    Point()
    { x=0;
      y=0;
    }
    Point(int x1, int y1)
    { x=x1;
      y=y1;
    }
    void print()
    {
      cout<<x<<" "<<y<<endl;
    }
};

int main()
{
  Point P1, P2(10,20);
  P1.print();
  P2.print();
  Point *ptr = new Point(5,10);
  ptr->print();
  return 0;
}
```

O/P:
0 0
10 20
5 10

# Initializer list

```
class Point
{ private:
    int x, y;
  public:
    Point() : x(0), y(0)
    {
    }
    Point(int x1, int y1) : x(x1), y(y1)
    {
    }
};
```

# Copy Constructor

```
Class Test {
    int *ptr;
public:
    Test(int x)
    {
        ptr = new int(x);
    }
    void set(int x)
    { *ptr = x;
    }
    void print()
    {
        cout << *ptr << " ";
    }
};
int main()
{
    Test t1(10);
    Test t2(t1);   // Test t2 = t1;
    t2.set(20);
    t1.print();
    t2.print();
}
```

Shallow copy

O/P:

20 20

```
class Test {
    int *ptr;
public:
    Test(int x)
    { ptr = new int(x);
    }
    Test(const Test &t)
    { int val = *(t.ptr);
      ptr = new int(val);
    }
    void set(int x)
    { *ptr = x;
    }
    void print()
    {
        cout << *ptr << " ";
    }
};
int main() {
    Test t1(10);
    Test t2(t1);
    t2.set(20);
    t1.print();
    t2.print();
}
```

copy Constructor

O/P:

10 20

Deep Copy

## Destructor

```
class Test
{
    public:
        Test() { cout << "constructor " << endl; }
        ~Test() { cout << "destructor" << endl; }
};
int main()
{
    Test t;
    return 0;
}
```

O/P: Constructor
Destructor

```
class Test
{ int x;
  public:
    Test(int i) : x(i)
    { cout << "cons " << x << endl;
    }
   ~Test()
    { cout << "des " << x << endl;
    }
};
int (main
```

```
int main()
{ Test t1(10);
  Test t2(20);
  return 0;
}
```

O/P:

```
Cons 10
cons 20
des 20
des 10
```

## Static Members

```
class Player {
  public:
    static int count;
    Player() { count++; }
    ~Player() { count--; }
};
int Player::count = 0;
int main()
{
  Player P1;
  cout << Player::count << " ";
  { Player P2;
    cout << Player::count << " ";
  }
  cout << Player::count << " ";
}
```

→ Scope resolution

O/P:

```
1 2 1
```

* Static functions only access static members however non-static functions can access static members.

## Inheritance

```
class Person {
    protected:
        String name;
        int id;

};

class Student : public Person {
    public:
        int getName()
        {
            return name;
        }

};
```

public: → protected and public of base class remain as they are.

private: → protected and public of base class become private.

protected: → protected and public of base class become protected

# Virtual functions

```
class Base
{ public:
    void print() { cout<< "Base\n"; }

};
class Derived : public Base
{
    public:
        void print() { cout<< " Derived \n"; }

};
int main()
 {
    Base b;
    Derived d;
    b.print();
    d.print();
    Base *ptr = &d;
    ptr -> print();
    return 0;
}
```

O/P:

```
Base
Derived
Base
```

```
class Base
{
    public:
        virtual void print() { cout<<"Base \n"; }
};
class Derived : public Base {
    public:
        void print() { cout<<"Derived \n"; }
};
int main()
{
    Base b;
    Derived d;
    b.print();
    d.print();
    Base *ptr = &d;
    ptr->print();
    return 0;
}
```

O/P:

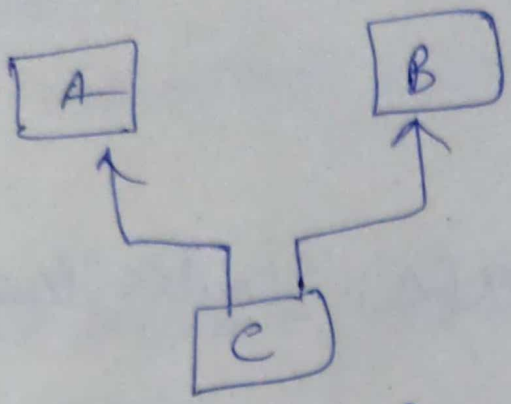
Base
Derived
Derived

# Inheritance Examples

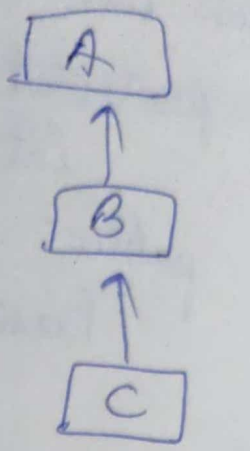
```
class Base {
    protected:
        int x;
    public:
        Base(int a) : x(a) { cout << "Base \n"; }

};
class Derived : public Base {
    private:
        int y;
    public:
        Derived(int a, int b) : Base(a), y(b) { cout << "Derived \n"; }
        void print() { cout << x << " " << y << endl; }

};
int main()
{
    Derived d(10,20);
    d.print();
    return 0;
}
```

O/P:
Base
Derived
10 20

# Multiple Inheritance



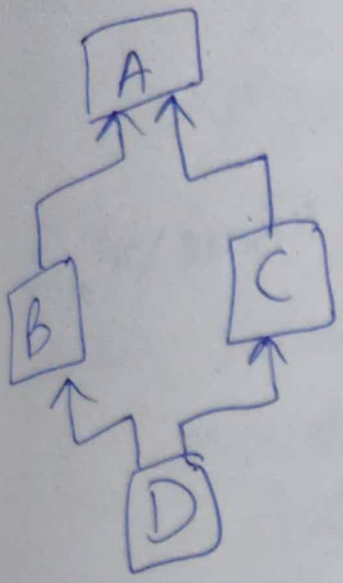

Multiple Inheritance




Multilevel

```
class A { };
class B { };
class C : public A, public B { };
int main {
    C obj;
    return 0;
}
```

# Diamond Problem




```
class A {
    public:
        int x = 10;
};
class B: virtual public A { };
class C: virtual public A { };
class D: public B, public C { };
int main()
{
    D d;
    cout << d.x;
    return 0;
}
```

O/P: Error

O/P:
10

↳ now no error after putting virtual keyword

```
class Base {
    private:
        int x;
    public:
        Base(int a) : x(a) { cout << "Base\n"; }
};
class Derived : public Base {
    private:
        int y;
    public:
        Derived(int a, int b) : Base(a), y(b) { cout << "Derived\n"; }
        void print() { cout << x << " " << y << endl; }
};
int main()
{
    Derived d(10, 20);
    d.print();
    return 0;
}
```

O/P:

Compiler error because of private in base class

```
class Base {
  public:
    int x;
    Base(int a): x(a) {cout<<"Base\n";}

};
class Derived : protected Base {
  private:
    int y;
  public:
    Derived(int a, int b): Base(a), y(b)
    { cout << "Derived \n"; }
    void print()
    { cout<< x<<" "<< y<<endl;
      }
};
int main()
{ Derived d(10,20);
  d.print();
  cout<<d.x;
  return 0;
}
```

O/P:

will not compile

→ Because public x in base class becomes protected in derived class, and it cannot be accessed outside the ~~class~~ derived class

# Operator Overloading

```
class Complex {
  private:
    int real, imag;
  public:
    Complex(int r=0, int i=0): real(r), imag(i){ }
    Complex operator + (Complex const &obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print()
    {
     cout<<real<<"+i"<<imag<<endl;
    }
};
int main()
{
    Complex C1(10,5), C2(2,4);
    Complex C3 = C1 + C2; // C1.operator+(C2)
    C3.print();
    return 0;
}
```

O/P:

12 + i9

. , :: , ?: and sizeof cannot be overloaded

# Friend functions and classes

→ Accesses private and protected members of other class.

```
class Employee;
class Printer {
    public:
        void printEmp(const Employee &e);
};
class Employee {
    private:
        int id;
        string name;
    public:
        friend void Printer::printEmp(const Employee &e);
        Employee(int i, string n): id(i), name(n){}
};
void Printer::printEmp(const Employee &e)
{
  cout<<e.id<<" "<<e.name<<" ";
}
int main()
{
    Printer p;
    Employee e(101, "ABC");
    p.printEmp(e);
    return 0;
}
```

O/P:

101 ABC