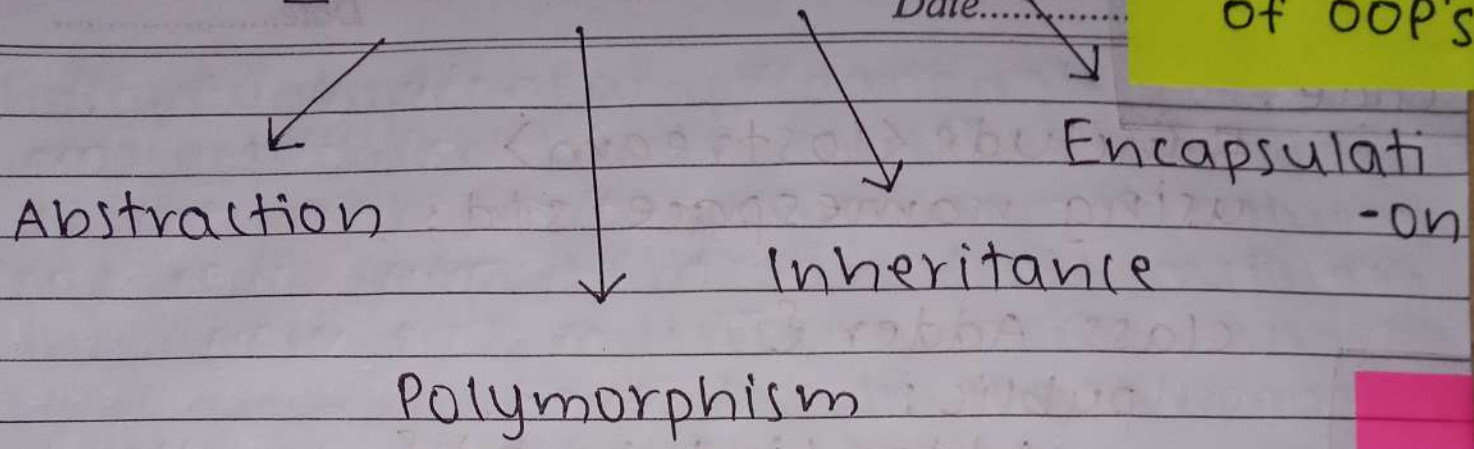
①

[OOPS Implementation]

Date.....

Implemento of OOP's



Abstraction :- Data abstraction refers to providing only essential information to the outside world & hiding their background details, i.e., to represent the needed information in program without presenting the details.

In C++, classes provides great level of Data abstraction. They provide sufficie -nt public methods to the outside world to play with the functionality of the object & to manipulate object data, i.e., state without actually knowing how class has been impleme -nted internally.

Any C++ program where we implement a class with public & private members is an example of Data Abstraction.

Its main goal is to handle complexity by hiding unnecessary details from the user.

(2)

Date.....

code:

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

class Adder{
    public:
        Adder(int i = 0){
            total = i;
        }
        void addNum(int number){
            total += number;
        }
        int getTotal(){
            return total;
        }
    private:
        int total;
};

int main(){
    Adder a;
    a.addNum(10);
    a.addNum(20);
    a.addNum(30);
    cout << "Total " << a.getTotal();
    return 0;
}
```

constructor

Date.....

Encapsulation :- Encapsulation is the concept that binds together the data & functions that manipulate the data, and that keeps both safe from outside interference & misuse.

Data Encapsulation led to the important concept of "Data Hiding".

Data Encapsulation is a mechanism of binding the data, and the functions that use them.

Data Abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsula-tion & data hiding through the creati-on of user defined types, called classes.

Code:

```
class Box {
  public:
    double getVolume (void) {
      return length * breadth *
                    height;
    }
  private:
    double length;
    double breadth;
    double height;
};
```

(4)

Date.....

The variables length, breadth & height are private. This means that they can be accessed only by other members of the Box class & not by any other part of your program. This is one way encapsulation is achieved.
Making one class a friend of another exposes the implementation details & reduces encapsulation

Date.....

Inheritance :- Inheritance allows to define a class in terms of another class, which makes it easier to create & maintain an application. This also prov-ide an opportunity to reuse the code functionality & fast implementation time.

In C++, we can reuse a class & also add additional features to it. Reusing classes can save time & money.

Reusing already tested & debugged class will save a lot of efforts of developi-ng & bugs checking the same thing again.

we can reuse the properties of an existing class by inheriting from it.

The existing class is called Base Class, and the new class is called as Derived class.

A class can be derived from more than one classes, which means it can inherit data & functions from multiple base classes.

Date.....

code:

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

class Shape{
  public:
    void setwidth(int w){
      width = w;
    }
    void setHeight(int h){
      height = h;
    }
  protected:
    int width;
    int height;
};

class Rectangle : public Shape{
  public:
    int getArea(){
      return width * height;
    }
};

int main(){
  Rectangle Rect;
  Rect.setWidth(5);
  Rect.setHeight(7);
  cout << "Total area: " <<
      Rect.getArea() << endl;
  return 0;
}
```

Date.....

A derived class can access all the non private members of its Base class. Thus, Base class members that should not be accessible to the member functions of derived classes should be declared private in the Base class. Table to summarize the access types according to who can access them :-

Access	Public	Protected	Private
Same class	Yes	Yes	Yes
Derived class	Yes	Yes	NO
Outside class	Yes	NO	NO

A derived class inherits all base class methods with the following exceptions:

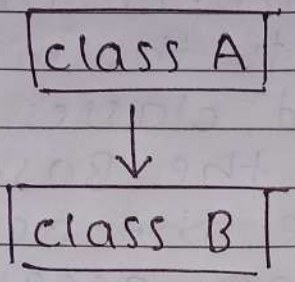
i) Constructors, Destructors & Copy constructors of the Base class.
ii) Overloaded operators of the Base class.
iii) The friend functions of the Base class.

Types of Inheritance :-

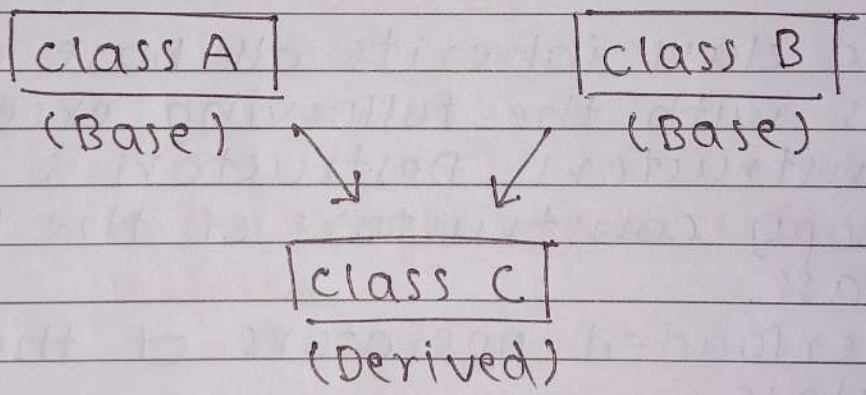
i) Single Inheritance
ii) Multilevel Inheritance
iii) Multiple Inheritance
iv) Hierarchical Inheritance
v) Hybrid Inheritance

Date.....

* Single Inheritance : A derived class with only one Base Class.



* Multiple Inheritance : Here, a class can inherit from more than one class, i.e., a derived class with more than one Base class.

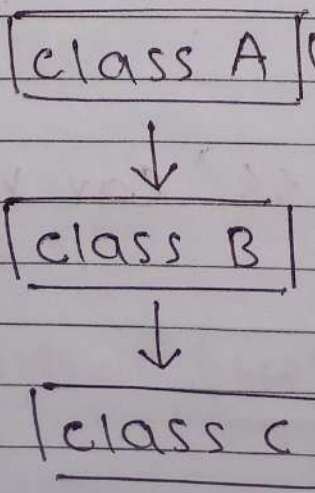



```
#include<iostream>
using namespace std;
class vehicle
{
  public:
    vehicle()
    {
      cout<<" vehicle ";
    }
};
```

Date.....

```
class Fourwheeler
{
  public:
    Fourwheeler()
    {
      cout << " 4 wheeler ";
    }
};

class car: public vehicle, public
                          Fourwheeler
{
};

int main()
{
   car obj;
   return 0;
}
```

* Multilevel Inheritance : Here, a derived class is created from another derived class.



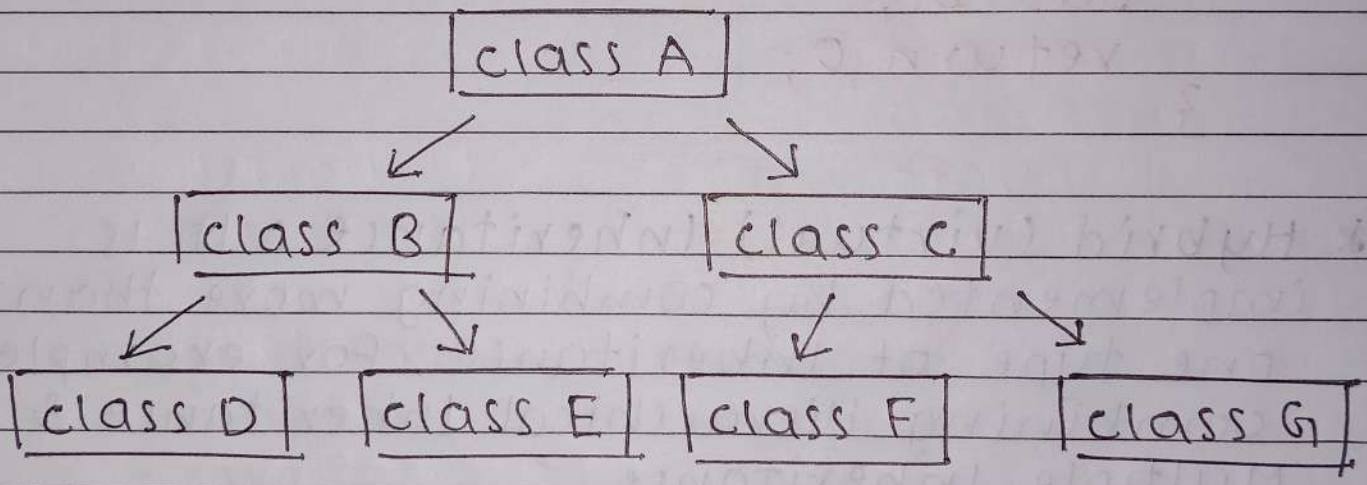
Date.....

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

class vehicle
{
  public:
    vehicle()
    {
      cout << "This is vehicle";
    }
};

class fourwheeler : public vehicle
{
  public:
    fourwheeler()
    {
      cout << "4 wheels are good";
    }
};

class car : public fourwheeler
{
  public:
    car()
    {
      cout << "Car has 4 wheels";
    }
};
```

Date.....

```
int main()
{
    car obj;
    return 0;
}
```

* Hierarchical Inheritance: Here, more than one subclass is inherited from a single Base class, i.e., more than one derived class is created from a single Base class.

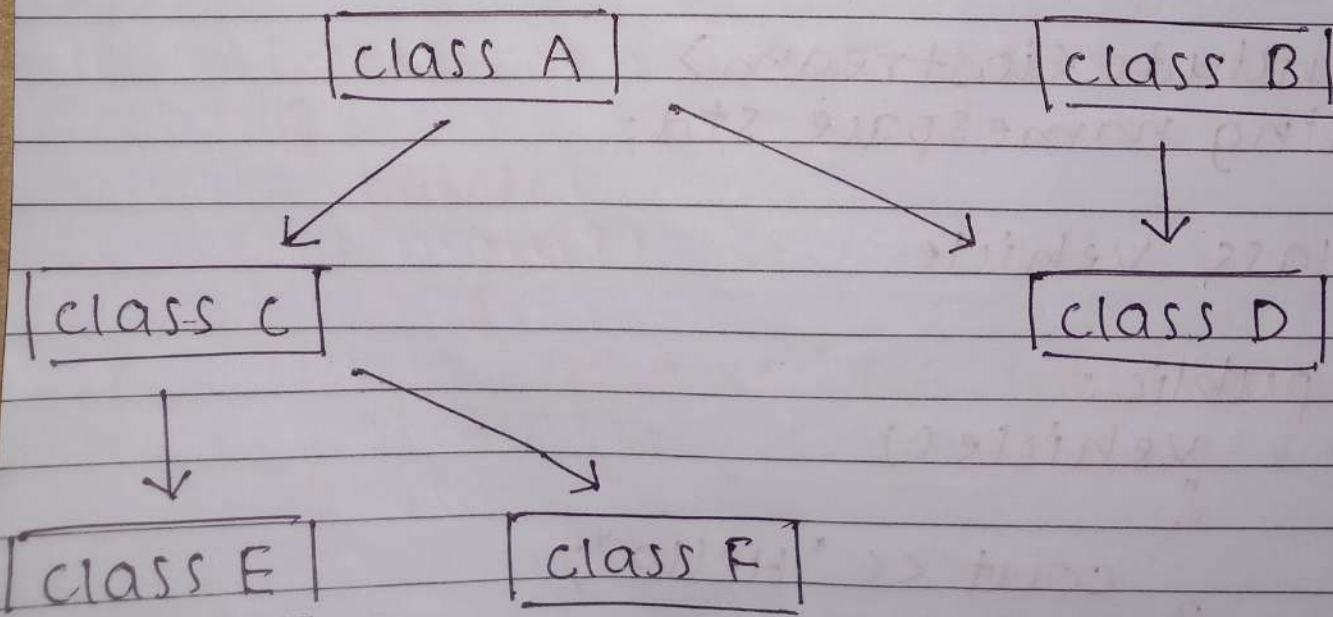



```
#include<iostream>
using namespace std;

class vehicle
{
  public:
    vehicle()
    {
      cout << "Hello";
    }
};
```

Date.....

```
class Car : public Vehicle
{
};

class Bus : public Vehicle
{
};

int main()
{
   Car c1;
   Bus b1;
   return 0;
}
```

* Hybrid (virtual) Inheritance : It is implemented by combining more than one type of inheritance. For example, combining Hierarchical Inheritance & Multiple Inheritance.



(13)

Date.....

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

class A
{
   protected:
      int a;
   public:
      void get_a()
      {
         cout<<"Enter a: ";
         cin>>a;
      }
};

class B : public A
{
   protected:
      int b;
   public:
      void get_b()
      {
         cout<<"Enter b: ";
         cin>>b;
      }
};

class C
{
   protected:
      int c;
```

Date.....

```
    public:
        void get_c()
        {
            cout << " Enter c: ";
            cin >> c;
        }
};

class D: public B, public C
{
    protected:
        int d;
    public:
        void mul()
        {
            get_a();
            get_b();
            get_c();
            cout << "Multiplication of a,
                b, c is: " << a*b*c;
        }
};

int main()
{
    D d;
    d.mul();
    return 0;
}
```

Date.....

There is a special case of Hybrid Inheri-tance : Multipath Inheritance.

A derived class is two base classes & these two base classes have one common Base class is called multipath inheritance. Ambiguity can arise in this type of Inheritance.

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

class A
{
  public :
    int a;
};

class B : public A
{
  public:
    int b;
};

class C : public A
{
  public:
    int c;
};
```

Date.....

```
class D : public B, public C
{
  public:
    int d;
};

int main()
{
    D obj;
    obj.a = 10;
    obj.a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a: " << obj.a;
    cout << " b: " << obj.b;
    cout << " c: " << obj.c;
    cout << " d: " << obj.d;
}
```

} ambiguity

In above program, both class B & class C inherits class A, they both have a single copy of class A.
However, class D inherits both class B & class C, there class D has two copies of class A, one from class B and another from class C.

Date.....

If we need to access the data members of class A through the object of class D, then there is an ambiguity occurs because D has two copies of A, one from class B & another from class C, and compiler can't differentiate between two copies of class A in class D.

There are 2 ways to avoid Ambiguity:

(1) Avoiding ambiguity using the scope resolution operator :- Using the scope resolution operator we can manually specify the path from which data member 'a' will be accessed.
For example :-

```
int main()
{
    D obj;
    obj.B::a = 10;
    obj.C::a = 100;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;
    cout << " a from B : " << obj.B::a;
    cout << " a from c : " << obj.C::a;
    cout << " b : " << obj.b;
    cout << " c : " << obj.c;
    cout << " d : " << obj.d;
}
```

from '::' this scope resolution operator, we can avoid ambiguity.

Date.....

After using the scope resolution operator '::', still there are two copies of class A in class D.

(2) Avoiding ambiguity using the virtual Base Class:-

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

class A
{
  public:
    int a;
};

class B : virtual public A
{
  public:
    int b;
};

class C : virtual public A
{
  public:
    int c;
};
```

Date.....

```
class D : public B, public C
{
  public:
    int d;
};

int main()
{
    D obj;
    obj.a = 10; // statement 1
    obj.a = 100; // statement 2
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "a: " << obj.a;
    cout << "b: " << obj.b;
    cout << "c: " << obj.c;
    cout << "d: " << obj.d;
}
```

In above example, class D has only one copy of class A, therefore "statement 2" will overwrite the value of 'a', given in "statement 1"

Date.....

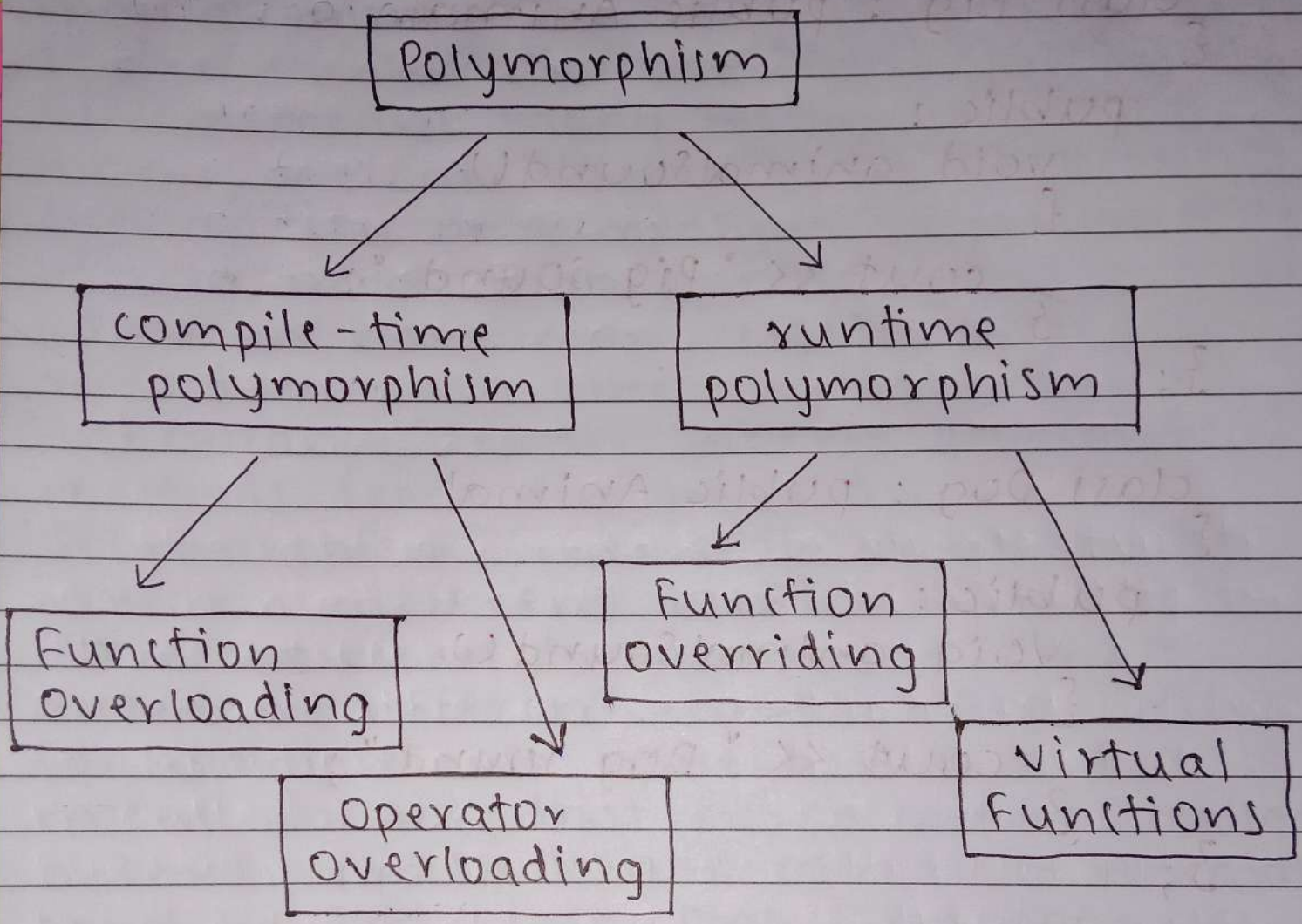
Polymorphism :- The term "Polymorphism" is the combination of "Poly" + "Morphs", which means many forms. In simple words, we can define polymorphism as the ability of a message to be display -ed in more than one form, it occurs when we have many classes that are related to each other by Inheritance. "Inheritance" let us inherit attribute & methods from another class. "Polymorphism" uses those methods to perform different tasks. This allows us to perform a single action in different ways. For example, we have a Base class called Animal, that has a method called animalSound(). Derived classes of Animal could be Pigs, Cats, Dogs, Birds, etc., & they also have their own implementa -tion of an animal sound

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

class Animal
{
  public:
    void animalSound()
    {
      cout<<"animal make sound";
    }
};
```

Date.....

```
class Pig : public Animal
{
  public:
    void animalSound()
    {
      cout << "Pig sound.";
    }
};

class Dog : public Animal
{
  public:
    void animalsound()
    {
      cout << "Dog sound";
    }
};

int main()
{
    Animal a;
    Pig p;
    Dog d;

    a.animalSound();
    p.animalSound();
    d.animalSound();

    return 0;
}
```

Date.....



* compile-time polymorphism : This type of polymorphism is achieved by Function Overloading or Operator Overlo-ading.

→ Function Overloading : when there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading.

Date.....

Functions can be overloaded by changing the number of argumen -ts or changing the type of arguments. This the feature of OOPs providing many functions to have the same name but distinct parameters when numero -us tasks are listed under one function name.

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

class GeeKs
{
  public:
    void func(int x)
    {
      cout << "value of x is " << x;
    }

    void func(double x)
    {
      cout << "value of x is " << x;
    }

    void func(int x, int y)
    {
      cout << "value of x and y is " <<
                                x << y;
    }
};
```

Date.....

```
int main()
{
    Geeks g;
    g.func(7);
    g.func(9.132);
    g.func(85, 64);

    return 0;
}
```

In the above program, a single funcTi -on named func() acts differently in three different situations.
Based on the argument passed during a function call, a particular func() is called.
Function Overloading is called compile time polymorphism because the compiler knows which function to execute before the program is compiler/ executed.

Date.....

→ Operator overloading : C++ has the ability to provide the operato -rs with the special meaning for a user defined data type, this ability is knows as operator overlo -ading.

operator overloading means defining additional tasks to operators without changing its actual meaning, we do this by using operator function.

The advantage of operator overloa -ding is to perform different operations on the same operand, & it gives more functionality / flexibi -lity of using operators with our user defined data types.

Let suppose, we've created a class named "complex". In this class, we've 2 data members, i.e., 'real' & 'imag'.
If we create objects of this complex class, i.e., c1(5, 4) & c2(2, 4) & if we try to add these two objects then compiler gives the error, because directly we can add user defined datatypes, for this we have to do operator overloading using "operator" keyword.

Date.....

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

class complex
{
  private:
    int real, imag;
  public:
    complex()
    {
      real = 0;
      imag = 0;
    }
    complex(int r, int i)
    {
      real = r;
      imag = i;
    }
    void print()
    {
      cout << real << " + " << imag << "i";
    }
};

int main()
{
  complex c1(2, 4);
  complex c2(3, 7);
  complex c3;
  c3 = c1 + c2;
  return 0;
}
```

this statement will give error because '+' operator can't directly add user defined objects. Here, we use operator overloading.

Spiral

Date.....

'+' operator does not work on user defined objects directly. So, we overload '+' operator in a way that it will know that what it has to do with this '+' operator & with the class objects.

In order to perform operator overloading, we have to write special function, i.e.;

```
Complex operator +(complex c)
{
    complex temp;
    temp.real = real + c.real;
    temp.imag = imag + c.imag;
    return temp;
}
```

Now, whenever the '+' operator is used with the object of the class, then automatically the overloaded function is called & do the functionality inside that function.

Date.....

code :-

```
#include<iostream>
using namespace std;
class complex{
  private:
    int real, imag;
  public:
    complex(){
      real = 0;
      imag = 0;
    }
    complex(int r, int i){
      real = r;
      imag = i;
    }
    void print(){
      cout << real << "+" <<
        imag << "i";
    }
    complex operator + (complex c)
    {
      complex temp;
      temp.real = real + c.real;
      temp.imag = imag + c.imag;
      return temp;
    }
};
int main(){
  complex c1(5, 4);
  complex c2(2, 5);
  complex c3(1, 1);
  complex c4;
```

Date.....

```
    c4 = c1 + c2 + c3;
    c4.print();
    return 0;
}
```

The advantage of operator overloading is that it gives us more flexibility of using operators with our user defined data type.
So, instead of creating functions for adding & subtracting, we can directly use these operators with our user defined objects then we have to perfo -rm operator overloading.

* Compile Time Polymorphism is also called Early Binding or Static Binding or Static Polymorphism

Other names of compile time polymorphi -sm :-

i) Early Polymorphism
or
Early Binding.

ii) Static Polymorphism
or
Static Binding.

Date.....

* Run-time polymorphism : This type of Polymorphism is achieved by Function overriding & virtual Function.

→ Function Overriding : If derived class defines same function as defined in its Base class, then it is known as Function Overridi -ng.
If we create an object of the derived class & call the member function which exists in both classes (Base & Derived), the member function of the derived class is invoked & the function of the Base class is ignored.
Function overriding enables us to provide specific implement of the function which is already provided by its Base class.

Date.....

```
class Base {
  public:
    void getData() {
      ——
      ——
    }
};

class Derived : public Base {
  public:
    void getData() {
      ——
      ——
    }
};

int main() {
  Derived dr;
  dr.getData;
}
```

This function is not called

Function call goes here

The indirect way to solve this problem is call the Base class function in Derived class function using the scope resolution '::' operator.

Date.....

```
class Base {
   public:
virtual void getData() {
        }
};

class Derived : public Base {
   public:
    void getData() {
        Base::getData();
    }
};

int main() {
   Derived dr;
   dr.getData();
}
```

Function call 2

Function call 1

this is the indirect way of calling Base class method from Derived class object.

Date.....

→ virtual Function : A virtual functi-on is declared by keyword "virtual". A virtual Function is the member function in the Base class. we can redefine it in the Derived class. It is a part of Runtime Polymorphi-sm. The declaration of virtual Function must be in the Base class by using the keyword "virtual" A virtual Function is not static.

The virtual Function helps the compi-ler to perform Dynamic Binding or Late Binding on the Function.

If it is necessary to use a single pointer to refer all the different classes's objects. This is because, we will have to create a pointer to the base class that refers to all the derived objects. But, when the Base class pointer contains the derived class address, the object always executes the Base class Function. For resolving this problem, we use the virtual Function.

when we declare a virtual Function, the compiler determines which Function to invoke at Runtime.

Date.....

Program execution without virtual function :-

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

class Add {
    int x = 5, y = 20;
    public:
        void display(){
            cout << "value of x is : " <<
                                    x + y;
        }
};

class subtract : public Add {
    int y = 10;
    int z = 30;
    public:
        void display(){
            cout << "value of y is :"
                                    y - z;
        }
};

int main(){
    Add *m; // Base class pointer
    subtract s; // child class object
    m = &s; // Base class pointer contain
    m -> display();          child class
    return 0;                Object address.
}
```

Spiral

Date.....

In previous program, we have created a Base class pointer & a child class object, i.e.,

```
Add *m; // Base class Pointer
Subtract s; // child class object
```

After this we assign child class object's address to Base class pointer variable,

```
m = &s; // child class object
        address assigned to
        Base class Pointer
        variable.
```

Now, when we call "display()" method through Base class Pointer variable,

```
m -> display();
```

Generally, this method call should call the child class display() method because m is containing the address of child class object, but this function call is calling the Base class display() method because m is the pointer variable of Base class.

So, the output of previous program is :-

value of x is : 25

Date.....

So, this problem is solved using "virtual function".
Virtual Function is declared using "Virtual" Keyword.

Program execution with Virtual Function :-

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

class Add {
    public:
        virtual void print () {
            int a = 20, b = 30;
            cout << "Base class action is :"
                << a + b;
        }
        void show() {
            cout << "Show Base Class";
        }
};

class Sub : public Add {
    public:
        void print () {
            int x = 20, y = 10;
            cout << "Derived class
                action is :" << x - y;
        }
        void show() {
            cout << "Show Derived.";
        }
};
```

Date.....

```
int main(){
    Add *aptr;
    sub s;
    aptr = &s;

    aptr -> print();
    aptr -> show();

    return 0;
}
```

In above program, we've created 2 public functions, i.e; print() & show(). print() function is a virtual function in Base class & show() function is a normal function.
In main() method, we've created a pointer variable of Base class & object of child class.
And we assign the child class address to Base class pointer, & now when we call virtual void print() method through Base class pointer then it will call Derived class print() method & not the Base class print() method because of "virtual" keyword. Similarly, we call show method through Base class pointer, then it will call Base class show method because show() method is not virtual function.

Date.....

So, the output of previous program is:

```
Derived class action is : 10
show Base class
```

* Pure virtual Function : when the function has no definition, we call such function as "DO Nothing Function" or "Pure virtual Function".
The declaration of this function happens in the Base class with no definition.

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

class Animal {
  public:
    virtual void show() = 0;
};
      ↳ Pure virtual function
          Declaration.

class Man : public Animal {
  public:
    void show(){
      cout << " Man is part of
          animal.";
    }
};
```

Date.....

```
int main() {
    Animal *aptr;
    Man m;
    aptr = &m;
    aptr -> show();
    return 0;
}
```

output of above code is :-

Man is part of animal.

Date.....

* Runtime Polymorphism example using two derived classes :-

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

class Polygon {
   public:
      virtual void show() {
         cout << "It is polygon";
      }
};

class Hexagon : public Polygon {
   public:
      void show () {
         cout << "Hexagon is 6 slide
            Polygon.";
      }
};

class Pentagon: public Polygon {
   public:
      void show() {
         cout << "Pentagon is 6 slide
            Polygon";
      }
};
```

Date.....

```
int main() {
    Polygon *p;
    Hexagon h;
    Pentagon ph;
     p = &h;
    p -> show();
     p = &ph;
    p -> show();
    return 0;
}
```

Output of above program is :-

```
Hexagon is 6 slide polygon.
Pentagon is 6 slide polygon.
```

Date.....

Abstract class in C++ :-

An abstract class in C++ is a class that has atleast one pure virtual function (i.e., a function that has no definition).

The classes inheriting the abstract class must provide a definition for the pure virtual function otherwise, the subclass would become an abstract class itself.

Abstract classes are essential to providing an abstraction to the code to make it reusable & extendable.

```
class Shape {
    public:
        virtual void calArea() = 0;
};
```

It doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual function

* The above class "shape" is an abstract class.

we cannot create objects of an abstract class. However, we can derive classes from them & use their data members & member functions (except pure virtual functions).

Date.....

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

class shape{
   protected:
      float dimension;
   public:
      void getDimension(){
         cin >> dimension;
      }
      virtual float calArea() = 0;
};

class Square : public shape{
   public:
      float calArea(){
         return dimension *
                        dimension;
      }
};

class Circle : public shape{
   public:
      float calArea(){
         return 3.14 * dimension *
                        dimension;
      }
};
```

Date.....

```
int main() {
    Square sq;
    Circle cir;

    cout << "Enter the length of the
        square:";
    sq.getDimension();
    cout << "Area of square:" <<
                    sq.calArea();

    cout << "Enter radius of the circle";
    cir.getDimension();
    cout << "Area of circle:" <<
                    cir.calArea();

    return 0;
}
```

In above program, "virtual float calArea() = 0" inside the shape class is a pure virtual function.
Thats why we must provide the implementation of "calArea()" in both of our derived classes or else we will get an error.