1. **Polymorphism:-**
   The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
   Exapmle:- A real-life example of polymorphism is a person who at the same time can have different characteristics. Like a man at the same time is a father, a husband and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism.

There are 2 types of polymorphism:

- **Compile time polymorphism**
- **Run time polymorphism**

Compile time / Static / Early Binding polymorphism

In C++ programming you can achieve compile time polymorphism in two way, which is given below;

Connecting the function call to the function body is called Binding. When it is done before the program is run or at compile time, its called **Early Binding** or **Static Binding** or **Compile-time Binding.**

- **Function Overloading**
- **Operator Overloading**
- **Function Overriding**

1. **Function Overloading:-**
   Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading.

```
class Addition
{
public:
        void sum(int a, int b)
        {
                cout<<a+b;
        }
        void sum(int a, int b, int c)
        {
                cout<<a+b+c;
        }
};
int main()
```

```
{
        Addition obj;
        obj.sum(10, 20);
        cout<<endl;
        obj.sum(10, 20, 30);
        return 0;
}
```

In the above example, a single function named ***sum*** acts differently in two different situations, which is a property of polymorphism.

2.**Function overriding:-**

Function overriding, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes.

```
class A{
        public:
          void f1(){
            cout<<"A f1()";
          }
          void f2(){
                cout<<"A f2()";
          }
};
class B : public A{
        public:
                void f1(){
                        cout<<"B f2()"; //method overriding
                }
                void f2(int x){
                        cout<<"A f2()"; //method hiding
                }

};
int main(){
```

```
    B b;
    b.f1();//B
    b.A :: f1(); // A
    a.f2();//error
    return 0;
}
```

Points to remember for for Function Overriding in C++

- Inheritance should be there. Function overriding cannot be done within a cl
- ass. For this we require a derived class and a base class.
- In C++, the base class member can be overridden by the derived class function with the same signature as the base class function.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.
- Method overriding is used to provide different implementations of a function so that a more specific behavior can be realized.
- Overriding is done in a child class for a method that is written in the parent class.
- It changes the existing functionality of the method..

3.Function Hiding:-

When you have the same function name inside derived class all the similar function name from the base class will be **hidden inside your derived class** and it doesn't matter parameter whether it is same or different.. you can't access base class function using derived class object.

In C++, function overloading is possible i.e., two or more functions from the same class can have the same name but different parameters. However, if a derived class redefines the base class member method then all the base class methods with the same name become hidden in the derived class.

Even if the signature of the derived class method is different, all the overloaded methods in the base class become hidden. For example, in the following program, Derived::f1(int) makes both Base::f1(char,int) and Base::fun(int) hidden.

```
class A{
    public:
      void f1(char a,int b){
        cout<<"A f1()";
      }
      void f1(int b){
        cout<<"A f1()";
      }
};
class B : public A{
    public:
        void f1(int a){
            cout<<"B f1()";
```

```
        }
};
int main(){
        B b;
        b.f1('a',5); //error
        b.f1(5); // B
        b.A::f1('a',b); //A
        return 0;
}
```

To use base class method in method hiding :-

1. Use scope resolution operator.
2. Use **Base :: fun_name;**

```
class A{
        public:
          void f1(char a){
             cout<<"A f1()";
          }
};
class B : public A{
        public:
                using A :: f1; //f1 of A will become local
                void f1(int a){
                        cout<<"B f1()";
                }
};
int main(){
        B b;
        b.f1(5); //B
        b.f1('a'); //A
        return 0;
}
```

3. ## Operator Overloading:-

When an operator is overloaded with multiple jobs, it is known as operator overloading. It is a way to implement compile time polymorphism.any symbol can be used as function name

- if it is a valid opeartor in c language
- If it is preceded by operator keyword

You can not overload **sizeof and ?:** operator.

Without Operator Overloading

```
class Complex{
   private:
      int a,b;
   public:
     void setData(int x,int y){
          a=x; b=y;
      }
      void showData(int x,int y){
          cout<<"a"<<a<<"b"<<b;
      }
      Complex add(Complex c){
        Complex temp;
        temp.a = a + c.a;
        temp.b = b + c.b;
        return temp;
      }
 };

void main(){
complex c1,c2,c3;
c1.setData(3,4);
c2.setData(5,6);
c3=c1.add(c2);        //c3 = c1+c2;
c3.showData();       // here it is non premitive
}
```

With Operator Overloading

```
class Complex{
   private:
      int a,b;
   public:
     void setData(int x,int y){
          a=x; b=y;
      }
      void showData(int x,int y){
          cout<<"a"<<a<<"b"<<b;
      }
      Complex operator +(Complex c){ //we can use + with operator only
        Complex temp;
        temp.a = a + c.a;
        temp.b = b + c.b;
        return temp;
      }
 };

void main(){
complex c1,c2,c3;
c1.setData(3,4);
c2.setData(5,6);
c3 = c1+c2;    // c3 = c1.operator+(c2)
c3.showData();
```

```
}
```

operator overloading of unary operator

```
class Complex{
    private:
        int a,b;
    public:
       void setData(int x,int y){
            a=x; b=y;
        }
        void showData(int x,int y){
            cout<<"a"<<a<<"b"<<b;
        }
        Complex operator -(){
          Complex temp;
          temp.a = -a;
          temp.b = -b;
          return temp;
       }
 };

void main(){
complex c1,c2;
c1.setData(3,4);
c2 = -c1;     //c2 = c1.operator-();
c3.showData();
}
```

operator overloading of ++(pre and post)

```
class Integer{
 private:
     int x;
 public:
    void setData(){ x=a;}
    void showDtata(){ cout<<"x="<<x;}
    Integer operator++(){
        Integer i;
        i.x=++x;   //pre increment
        return i;
    }
    Integer operator++(int){ //only show type
        Integer i;
        i.x=x++;  //post increment
        return i;
    }
}

void main(){
 Inter i1,i2;
```

```
 i1.setData(3);
 i1.sowData(); //3
 i2 = ++i1; // i2 = i1.operator++();
 i1.sowData(); //4
 i2.sowdata(); //4
 i2 = i1++;
 i1.sowData(); //5
 i2.sowdata(); //4
}
```

Runtime time / Dynamic / Late binding polymorphism

Dynamic polymorphism can be achieved using Virtual Functions in C++.

Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding / Dynamic Polymorphism** on this function. **Virtual** Keyword is used to make a member function of the base class Virtual.

Late Binding

In Late Binding function call is resolved at **runtime**. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called Dynamic Binding or Runtime Binding.

Base class pointer:-

Base class pointer can point to the object of any dascadent class ie child class **but converse is not true.**

Problem without Virtual Keyword

In this program you can see that even when we use the base class object to call the function of child class, it calls the base class function. This results in early binding which is not what we want. This problem can be resolved using the virtual keyword.

```
class Base

{

 public:

 void show()

 {

  cout << "Base class";

 }

};

class Derived:public Base
```

```
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
};

int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Early Binding Ocuurs base class
}
```

Using Virtual Keyword

We can make base class's methods virtual by using virtual keyword while declaring them. Virtual keyword will lead to Late Binding of that method. In this program given below you can see that we have made the base class function show as virtual by using the virtual keyword. Now in the main function we are called the derived class function using the base class pointer object. this is known as late binding or dynamic polymorphism

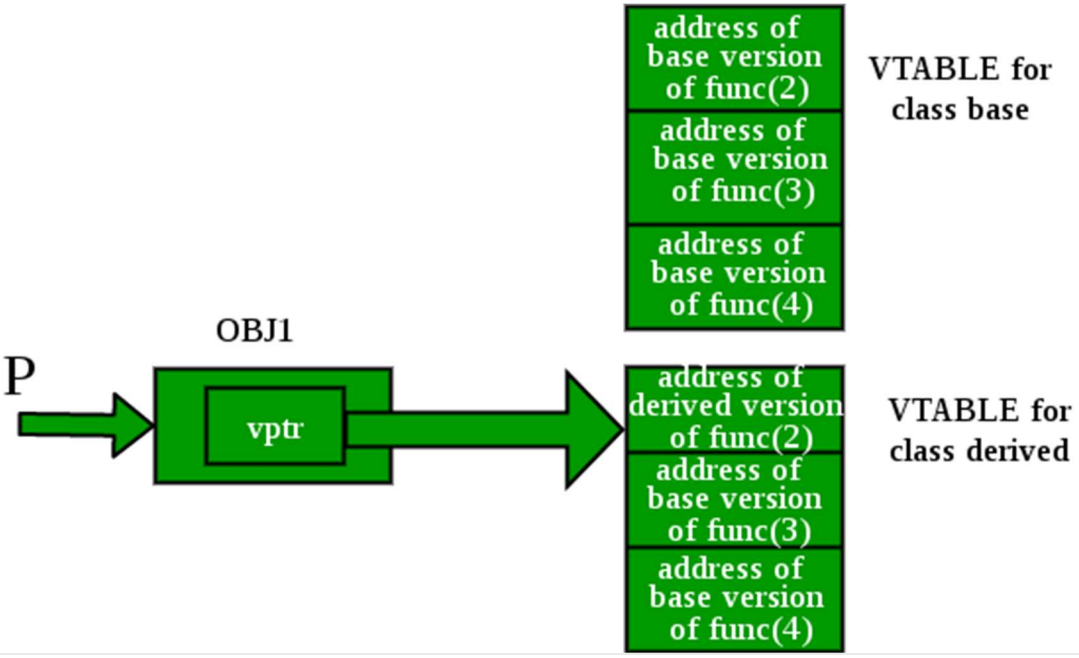
```
#include<iostream>
using namespace std;
class Base
{
 public:
 virtual void show()
 {
  cout << "Base class";
 }
```

```
};

class Derived:public Base

{

 public:

 void show()

 {

  cout << "Derived Class";

 }

};



int main()

{

 Base* b;        //Base class pointer

 Derived d;      //Derived class object

 b = &d;

 b->show();      //Late Binding Ocuurs

}
```

Working of virtual functions:-

1. **virtual pointer (VPTR):-** If object of that class is created then a **virtual pointer (VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

    2. **VTABLE:-** Irrespective of object is created or not, class contains as a member **a static array of function pointers called VTABLE**. Cells of this table store the address of each virtual function contained in that class.



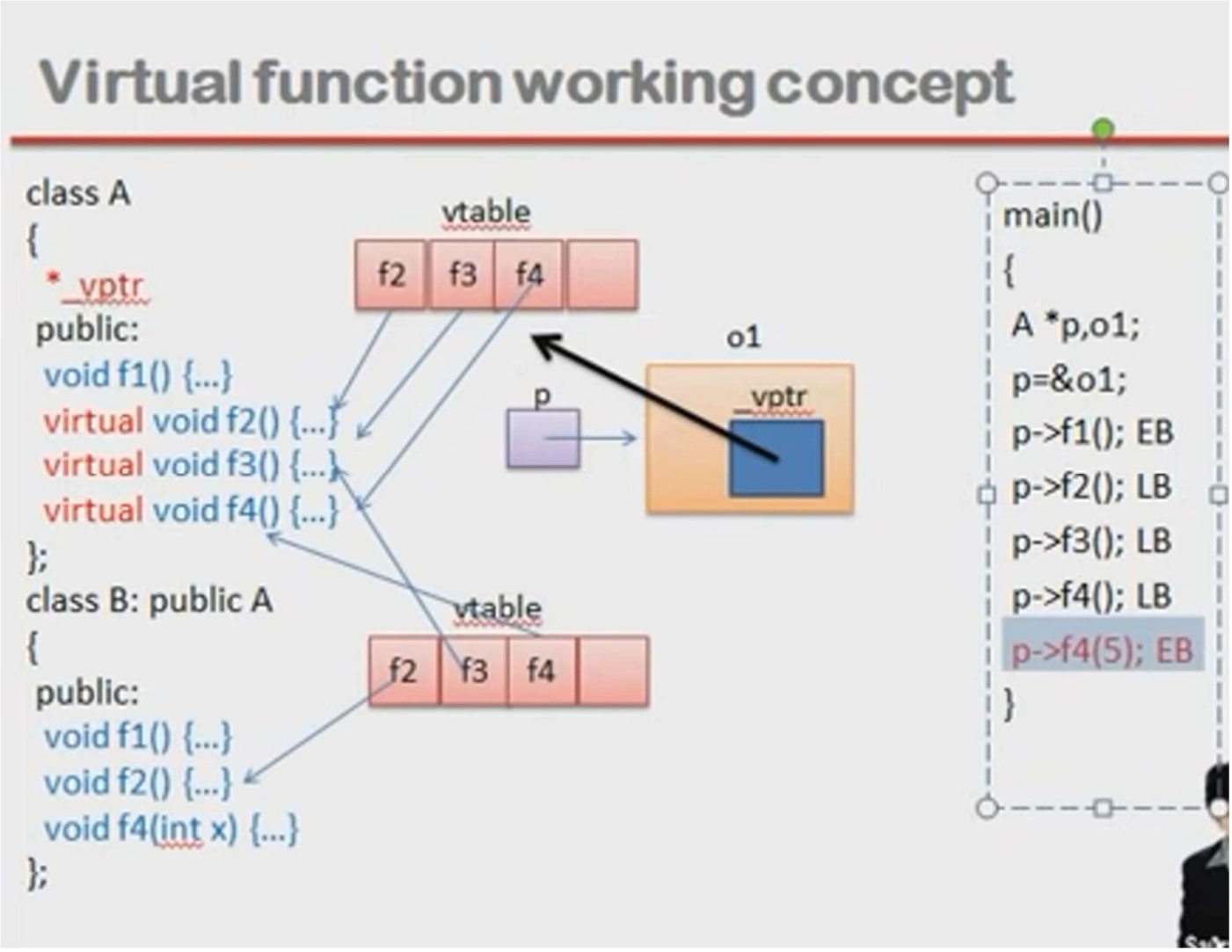

```
class base {

public:

	void fun_1() { cout << "base-1\n"; }

	virtual void fun_2() { cout << "base-2\n"; }

	virtual void fun_3() { cout << "base-3\n"; }

	virtual void fun_4() { cout << "base-4\n"; }

};

class derived : public base {

public:

	void fun_1() { cout << "derived-1\n"; }

	void fun_2() { cout << "derived-2\n"; }

	void fun_4(int x) { cout << "derived-4\n"; }

};

int main()

{

	base *p;

	derived obj1;
```

```
        p = &obj1;


        // Early binding because fun1() is non-virtual

        // in base

        p->fun_1();


        // Late binding (RTP)

        p->fun_2();


        // Late binding (RTP)

        p->fun_3();


        // Late binding (RTP)

        p->fun_4(); //Early binding but this function call is illegal
(produces error) because pointer is of base type and function is of
derived class   p->fun_4(5);
```

```
base-1

derived-2

base-3

base-4
```

Explanation: Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.
Similar concept of **Late and Early Binding** is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.
NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototypes of both the functions are different.

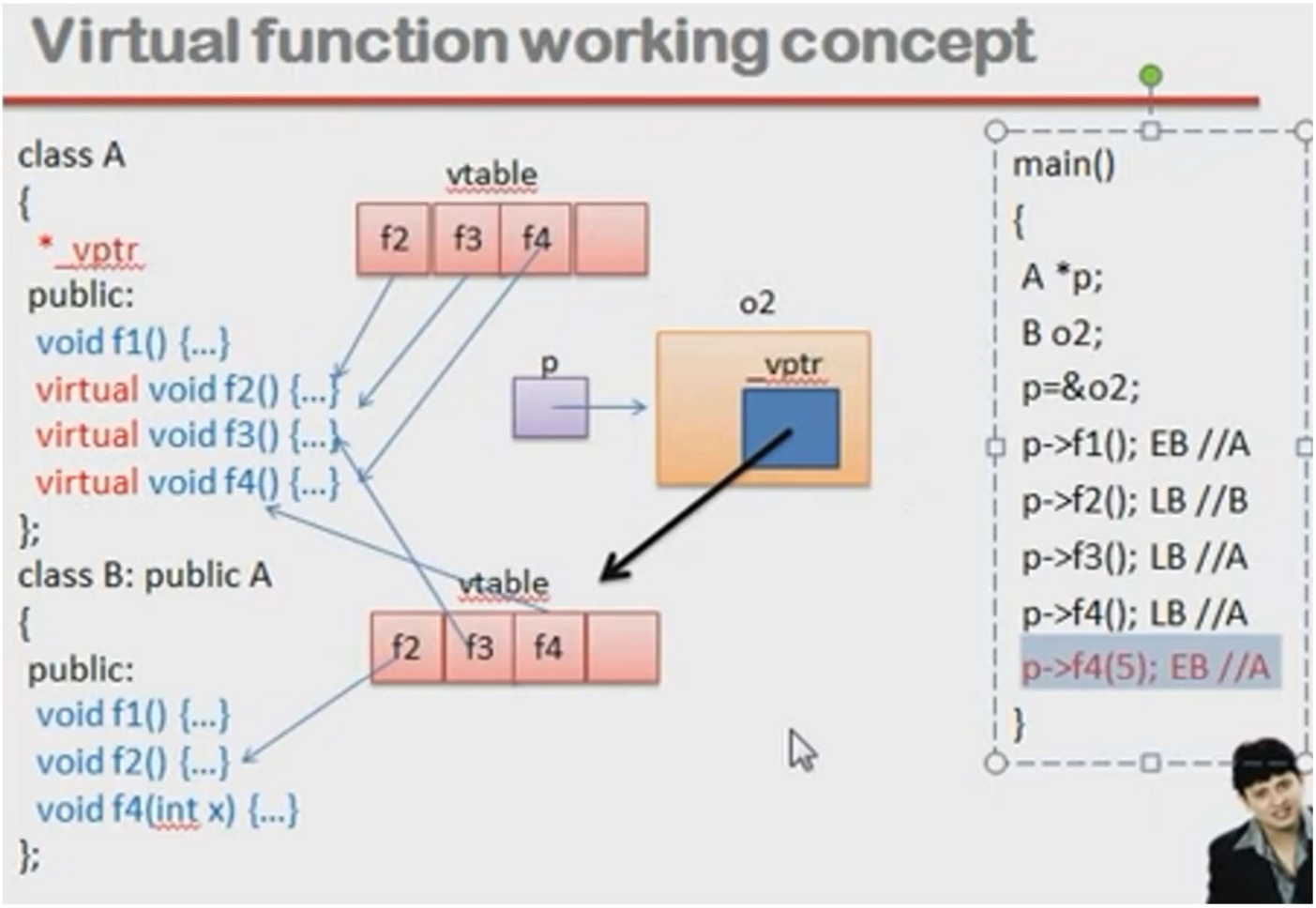
O1 class object



2.O2 class Object:-

1. write early binding or late binding by seeing the base ie parent class.

2.if child class not contain fn then parent eun in early binding.



Pure Virtual Function :-

A do nothing function is virtual function. Apan do nothing function ko virtual isiliye bana rahe h becz isko **early binding k through** access na kar paye partially implemented h. without virtual error ayegi.

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

```
class A{
        public:
          virtual void f1(char a)=0; //Pure virtual Funcion
};
class B : public A{
        public:
                void f1(char a){ // we cant use return type as int
                }
};
int main(){
```

```
    B b;
    return 0;
}
```

Abstract Class:-

A class containing pure virtual function is an abstract class. And we cannot instantiate abstract class.

Some Interesting Facts:

1) *A class is abstract* ***if it has at least one pure virtual function.***
In the following example, Test is an abstract class because it has a pure virtual function show().

2) *If we do not override the pure virtual function in derived class, then derived class also* ***becomes abstract class.***

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
class Derived : public Base { };
int main(void)
{
Derived d; //error
return 0;
}
```

3). *An abstract class can have* ***constructors.***

Abstract base classes

Classes which have pure virtual functions(such as the one in the above example) are known as **Abstract base classes** or simply **Abstract class**. Abstract base classes **cannot be used to instantiate objects**. But an abstract base class is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities and perform **dynamic polymorphism**.