Polymorphism means **many forms**. It is an object-oriented programming concept that refers to the ability of a variable, function, or object to take on multiple forms, which are when the behavior of the same object or function is different in different contexts.

Polymorphism can occur within the class and when multiple classes are related by inheritance.

What is Polymorphism in C++?

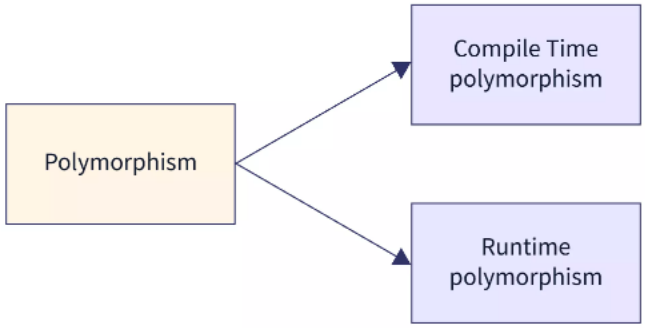
Polymorphism word is the combination of "poly," which means many + "morphs," which means forms, which together means many forms. Polymorphism in C++ is when the behavior of the same object or function is different in different contexts. Let's take a real-world example of the word right can mean different things in a different context.

- I was right. In the above sentence, the word **right** means "correct".
- Please take a right turn. The word **right** refers to the "right direction" in the above sentence.

Types of Polymorphism in C++

Following are the **two types** of polymorphism in C++ :

1. **Compile Time Polymorphism**
2. **Runtime Polymorphism**



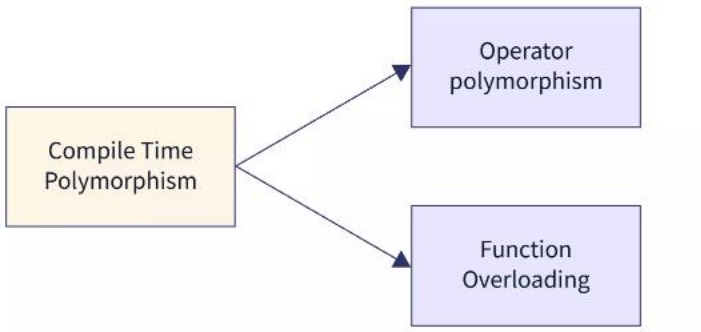
Compile Time Polymorphism

Compile-time polymorphism is done by overloading an **operator or function.** It is also known as "static" or "early binding".

Why is it called compile-time polymorphism?

Overloaded functions are called by comparing the data types and number of parameters. This type of information is available to the compiler at the compile time. Thus, the suitable function to be called will be chosen by the C++ compiler at compilation time.

There are the following types of compile-time polymorphism in C++ :



Function Overloading

When we have two functions with the same name but different parameters, different functions are called depending on the number and data types of parameters. This is known as function overloading

Function Overloading can be achieved through the following two cases:

- The names of the functions and return types are the same but differ in the type of arguments.
- The name of the functions and return types are the same, but they differ in the number of arguments.

Let's understand with an example :

```
#include<iostream.h>
#include<conio.h>
class Temp
{
   private:
      int x ;
      double x1;
   public:
     temp()
     {
          x=10;
          x1=10.1;
```

```
    }
    void add(int y)
    {
cout<< "Value of x + y is: " << x + y <<endl;
    }
    // Differ in the type of argument.
    void add(double d)
    {
cout<< "Value of x1 + d is: " << x1 + d <<endl;
    }
    // Differ in the number of arguments.
    void add(int y, int z)
    {
cout<< "Value of x + y + z is: " << x + y + z <<endl;
    }
};
int main() {
   Temp t1;
   t1.add(10);
   t1.add(11.1);
   t1.add(12,13);
return 0;
}
```

Output :

```
Value of x + y is: 20
Value of x1 + d is: 21.2
Value of x + y + z is: 35
```

Explanation:

- In the above example, the add function is overloaded.
- t1 is the object of class Temp, t1.add(10). It will call void add(int y).
- t1.add(11.1), it will call void add(double d).
- t1.add(12,13), it will call void add(int y, int z).
- The overloaded functions are called by matching the type and number of arguments. Because this information is available at compile-time, the compiler selects the proper function based on its parameters.

Operator Overloading

When an operator is updated to be used for user-defined data types (objects etc.), this is known as operator overloading. To use operator overloading, at least one operand must be a user-defined data type.

Note:

- A user-defined type must be present in at least one operand.
- ".", "::", typeid, size, ".*", and C++'s single ternary operator, "?:", are among the operators that cannot be overloaded.

These are some of the operators that can be overloaded in C++ :

Arithmetic operators : - , + , / , * , % and -=, +=, /= , *= , %=

Boolean algebra : !=, ==, > ,< ,>=, <= , && ,||

Bit manipulation : &, |, ^ ,<< ,>> and |=, &= ,>>=, <<=, ^=

Memory management : new[], new, delete[], delete

Note:

- New and Delete operators can be overloaded globally, or they can be overloaded for specific classes. If these operators are overloaded using the member function for a class, they are overloaded only for that specific class.

Syntax :

```
#include<iostream.h>
#include<conio.h>
Class test
{
        Int x;
        Public:
                Test()
                {}
                Test(int a)
                {
                        X=a;
                }
                Test operator+(test t2)
                {
                        Test t3;
                        T3.x=x+t2.x;
                        Return(t3);
                }
                Void display()
                {
                        Cout<<x<<endl;
                }
```

```
};
Void main()
{
        Clrscr();
        Test t1,t2,t3;
        T1=test(2);
        T2=test(3);
        T3=t1+t2;
        T1.display();
        T2.display();
        Cout<<"sum:"<<endl;
        T3.display();
        Getch();
}
```

Runtime Polymorphism

Runtime polymorphism occurs when functions are resolved at runtime rather than compile time when a call to an **overridden** method is resolved dynamically at runtime rather than compile time. It's also known as **late binding** or **dynamic binding**.

Runtime polymorphism is achieved using a combination of **function overriding and virtual functions**.

The following sections will show an example of overriding with and without the virtual keyword.

Function Overriding without Virtual Keyword

Function Overriding is when more than one method of the derived class has the same name, the same number of parameters, and the same parameter type as of base class.

Pointer To Derived Class

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, single pointer variable can be made to point object belonging to different classes.

Syntax :

```
// Without virtual function.
#include <iostream.h>
class Polygon {
   protected:
         int height, width;
   public:
         void set_values(int x, int y)
         {
               width = x;
               height = y;
         }
   int area() {
      return 0;
   }
};
class Rectangle: public Polygon {
   public: int area() {
      return width * height;
   }
};
class Triangle: public Polygon {
   public: int area() {
      return (width * height / 2);
   }
};
int main() {
   Rectangle rect;
   Triangle trgl;
   Polygon poly;

   Polygon * ppoly1 = &rect;
   Polygon * ppoly2 = &trgl;
   Polygon * ppoly3 = & poly;

   ppoly1 ->set_values(10, 20);
   ppoly2 ->set_values(10, 20);
   ppoly3 ->set_values(10, 20);

cout<< "Area of Rectangle is: " << ppoly1 -> area() << '\n';
cout<< "Area of Triangle is: " << ppoly2 -> area() << '\n';
cout<< "Area of Polygon is: " << ppoly3 -> area() << '\n';
   return 0;
```

}

Output:

Area of Rectangle is: 0
Area of Triangle is: 0

Area of Polygon is: 0

Explanation:

- In this example, the width, height, and functions set values and area are shared by all three classes (Polygon, Rectangle, and Triangle).
- Three references to Polygon are declared in the main function (ppoly1, ppoly2, and ppoly3). These are given the addresses rect and trgl poly, which are objects of type Rectangle, Triangle, and Polygon, respectively. Rectangle and Triangle are Polygon-derived classes; therefore, such assignments are permitted.
- ppoly1 ->set_values (10, 20); is similar to rect.set_values (10, 20); and ppoly2 ->set_values (10, 20); is similar to trgl.set_values (10, 20);
- ppoly1 -> area() and ppoly1 -> area() is called this will call area function of the base class Polygon.
- This happens due to static linkage, which implies that the compiler only sets the call to the area() once in the base class.
- To overcome this problem, the concept of virtual pointers is introduced.

Function Overriding with Virtual Keyword

A virtual function in C++ is a base class member function. In a derived class, we can redefine it. The virtual function must be declared using the keyword virtual in the base class. A class that declares or inherits a virtual function is called a **polymorphic class**.

When we use the same function name in both base and derived classes, the function in base class is declared as virtual using keyword virtual preceding its normal declaration.

When a function is made virtual, C++ determines which function to use at run time based by making the based on the type of object pointer to by base pointer, rather than the type pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

Syntax :

```
// With virtual function
#include <iostream.h>
class Polygon {
  protected:
        int height, width;
  public:
        void set_values(int x, int y)
```

```
            {
      width = x;
      height = y;
   }
   virtual int area() {
      return 0;
   }
};
class Rectangle: public Polygon {
   public: int area() {
      return width * height;
   }
};
class Triangle: public Polygon {
   public: int area() {
      return (width * height / 2);
   }
};
int main() {
   Rectangle rect;
   Triangle trgl;
   Polygon poly;

   Polygon * ppoly1 = &rect;
   Polygon * ppoly2 = &trgl;
   Polygon * ppoly3 = & poly;

   ppoly1 ->set_values(10, 20);
   ppoly2 ->set_values(10, 20);
   ppoly3 ->set_values(10, 20);

cout<< "Area of Rectangle is: " << ppoly1 -> area() << '\n';
cout<< "Area of Triangle is: " << ppoly2 -> area() << '\n';
cout<< "Area of Polygon is: " << ppoly3 -> area() << '\n';
   return 0;
}
```

Output:

```
Area of Rectangle is: 200
Area of Triangle is: 100
Area of Polygon is: 0
```

Explanation:

- ppoly1 -> area() is called it will call area function of Rectangle class.

- ppoly2 -> area() is called it will call area function of Triangle class.
- This is because we used the virtual keyword to make the base class Polygon method area virtual in this code. The confusion over which function to call will be resolved at runtime, resulting in the desired outcome.

```
class Base
 {
public:
void display()
{
cout<<"\n Display base";
}
virtual void show()
{
cout<<"\n show base";
}
};
class Derived: public Base
{
public:
void display()
{
cout<<"\n Display Derived";
}
void show()
{
cout<<"\n show derived";
}
};
int main ()
{
clrscr();
Base B;
Derived D;
Base *bptr;
cout<<"\n bptr points to base \n";
bptr=&B;
bptr -> display();//call base version
bptr -> show();//call base version
cout<<"\n\n bptr points to derived \n";
```

Output:

```
bptr points to base
Display base Show base
bptr points to Derived
```

Rules for virtual function

1. The virtual function must be member of base class
2. They cannot be static members
3. They are accessed by using object pointers
4. Prototype of base class function & derived class must be same
5. Virtual function in base class must be defined even though it is not used
6. A virtual function can be friend function of another class
7. We could not have virtual constructor
8. If a virtual function is derived in base class, it need not be necessarily redefined in the derived class
9. Pointer object of base class can point to any object of derived class but reverse is not true
10. When a base pointer points to derived class, incrementing & decrementing it will not make it point to the next object of derived class

Pure Virtual Function

A **Pure Virtual Function** is a virtual function declared in the base class without definition. In simple words, it does nothing in the base class. But the derived class must provide an implementation for this function. Otherwise, we'll get a compilation error.

A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function

To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

If you have a pure virtual, the class becomes abstract. You cannot create an object of it.

```
class base
 {
public:
//pure virtual function virtual void show()=0;
};
class derived1 : public base
{
public: void show()
{
 void main()
{
base *b;
derived1 d1;
b = &d1;
b->show();
}
```

```
cout<<"\n Derived 1";
}
};
```

Output
Derived 1

Syntax :

```
#include <iostream.h>
class Base {
   public: virtual void func() = 0; // Pure Virtual Function
};
class Derived: public Base {
   public: void func() {
cout<< "Inside derived class" <<endl;
   }
};
int main() {
   Base * b1;
   b1 = new Derived();
   b1 ->func();
   return 0;
}
```

Output:

Inside derived class

Explanation:

- Base *b1; a pointer to an object of the base class is created.
- b1 = new Derived(), b1 is now an object of Derived class.
- b1 ->func() will call the func() of the Derived class because it was declared as a virtual function in the Base class.

To check what error we get if we don't provide the definition of func() in Derived, comment out the definition and re-compile the above code. We get the following error:

```
g++ /tmp/main.cpp
/tmp/main.cpp: In function 'int main()':
/tmp/main.cpp:19:22: error: invalid new-expression of abstract
   19 |     b1 = new Derived();
      |                      ^
/tmp/main.cpp:9:7: note:   because the following virtual funct
    9 | class Derived : public Base
      |       ^~~~~~~
/tmp/main.cpp:6:17: note:       'virtual void Base::func()'
    6 |     virtual void func() = 0;     // Pure Virtual Functio
      |                 ^~~~
```

Compile-Time Polymorphism Vs. Run-Time Polymorphism

Compile Time Polymorphism	Run-Time Polymorphism
At Compile time, which functions to be called is decided.	At Runtime, which function to be called is decided.
Also known as early or static binding	Also known as late or dynamic binding
It executes faster because the function is resolved at compilation time only.	It executes slower because the function is resolved at Run-time.
It is achieved through function and operator overloading	It is achieved through function overriding and virtual functions

Conclusion

- Polymorphism in C++ is when the behavior of the same object or function is different in different contexts.
- It is of two types: **Compile-time Polymorphism** and **Runtime Polymorphism**.
- In Compile Time Polymorphism, the function to be invoked is decided at the compile time only. It is achieved using a function or operator overloading.
- In Runtime Polymorphism, the function invoked is decided at the Run-time. It is achieved using function overriding and virtual functions.
- The virtual function must be declared using the keyword "virtual" in the base class.
- A Pure Virtual Function is a virtual function declared in the base class without definition.

Friend Function:=

Que :In which circumstances function can be made as friend? Write the advantage of friend function. Demonstrate one example of friend function.

- Friend function is a special type of function, which declares inside the class. Friend function can access the private, protected and public data of the class.
- A keyword friend is used before return type of the function declaration/prototype.
- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
Class class_name
{
   Friend data_type function_name(argument/s);        //syntax of friend function.
};
```

- In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function.
- The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It can not be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It can not access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

```
#include<iostream.h>
#include<conio.h>
Class sample
{
        Int a,b;
        Public:
                Void getdata()
                {
                        A=25;
                        B=40;
```

```
                }
                Friend float mean(sample s);
};
Float mean(sample s)
{
        Return float(s.a+s.b)/2.0;
}
Void main()
{
        Sample x;
        x.getdata();
        cout<<"mean value="<<mean(x)<<endl;
        getch();
}
```

USING FRIEND FUNCTION TO ADD DATA OBJECTS OF TWO DIFFERENT CLASSES

```
#include<iostream.h>
#include<conio.h>
Class abc;
Class xyz
{
        Int data;
        Public:
                Void getdata(int value)
                {
                        Data=value;
                }
                Friend void add(xyz,abc);
};
Class abc
{
        Int data;
        Public:
                Void getdata(int value)
                {
                        Data=value;
                }
                Friend void add(xyz,abc);
};
Void add(xyz o1,abc o2)
{
        Cout<<"sum of objects using  friend function:"<<o1.data+o2.data;
}
```

```
Void main()
{
        Xyz x;
        Abc a;
        x.getdata(5);
        a.getdata(50);
        add(x,a);
        getch();
}
```

SWAPPING PRIVATE DATA OF CLASSES

```
#include<iostream.h>
#include<conio.h>
class abc;
class xyz
{
        int data;
        public:
                void getdata(int value)
                {
                        data=value;
                }
                void display()
                {
                        cout<<data<<endl;
                }
                friend void add(xyz &,abc &);
};

class abc
{
        int data;
        public:
                void getdata(int value)
                {
                        data=value;
                }
                void display()
                {
                        cout<<data<<endl;
                }
                friend void add(xyz &,abc &);
};
```

```
void add(xyz & o1,abc & o2)
{
        int temp=o1.data;
        o1.data=o2.data;
        o2.data=temp;
}
void main()
{
        clrscr();
        xyz x;
        abc a;
        x.getdata(5);
        a.getdata(50);
        x.display();
        a.display();
        add(x,a);
        x.display();
        a.display();
        getch();
}
```

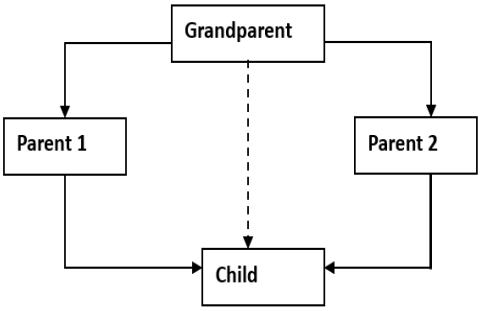
FRIEND CLASS

```
#include<iostream.h>
#include<conio.h>
class XYZ {
private:
  char ch;
  int num;
public:
        void data()
        {
                ch='a';
                num=9;
        }
  friend class ABC;
};
class ABC {
public:
  void disp(XYZ obj){
    cout<<obj.ch<<endl;
    cout<<obj.num<<endl;
  }
};
```

```
void main() {
  clrscr();
  ABC obj;
  XYZ obj2;
  obj2.data();
  obj.disp(obj2);
 getch();
}
```

Explain Virtual Base class with example

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.



→The child has two *direct base classes* 'parent 1' and 'parent 2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.
→Inheritance by the 'child' shown in fig. might pose some problem. All the public and protected members of 'grandparent' are inherited twice, first via 'parent 1' and again 'parent 2' This means 'child' would have duplicate sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

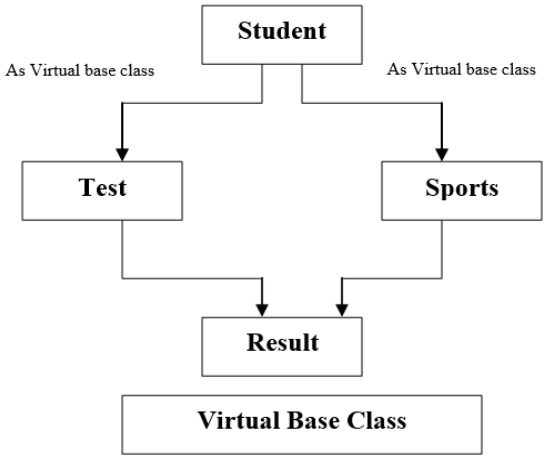
The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class.

Syntax

```
Class A  //grandparent
{

}
Class B1 : virtual public A //parent 1
{

}
Class B2 : public virtual A //parent 2
{

}
Class C : public B1, public B2 //child
{
                    //Only one copy of A
                   //will be inherited
}
```

Note: - The Keyword **virtual** and **public** may be use in either order.

For example: = Assume that class sports derives the roll_number from the class student. Then, the inheritance relationship will be shown in Fig.



Example:

```
class student
{
protected:
int roll_number;
public:
void get_number(int a)
{
roll_number =a;
}
void put_number(void)
{
out<<"Roll no"<<roll_number<<"\n";
}
};
class test: virtual public student
{
 public:
 protected:
float part1,part2;
 void get_mark(float x,float y)
{
part1=x; part2=y;
}
void put_marks()
{
cout<<"Marks obtained:"<<"part1="<<part1<<"\n"<<"part2="<<part2<<"\n";
}
};
class sports:public virtual student
{
 public:
 protected: float score;
 void get_score(float s)
{
score=s;
}
void put_score(void)
{
cout<<"sports:"<<score<<"\n";
}
};
class result: public test, public sports
{
```

```
public:
float total;
void display(void);
void result::display(void)
{
total=part1+part2+score;
put_number();
put_marks();
put_score();
cout<<"Total Score="<<total<<"\n";
}
};
int main()
{
clrscr();
result stu;
stu.get_number(123);
stu.get_mark(27.5,33.0);
stu.get_score(6.0);
stu.display();
return 0;
}
```

OUTPUT

```
Roll no 123
Marks obtained : part1=27.5
Part2=33
Sports=6
Total score = 66.5
```

Explain This Pointer

C++ uses unique keyword called this to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called.

This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions.

```
class abc
{
int a; public:
void display()
{
this->a=123; cout<<a;
```

```
}
};
main()
{
clrscr(); abc a; a.display();
getch();
}
```

→The Private variable 'a' can be used directly inside a member function, like a=123;
→We can also use the following statement to do same job:
this→a=123;
→Since C++ permits the use of shorthand form a=123, we have not been using pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Output:=
123

Explain Abstract Class

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class.

It is a design concept in program development and provides a base upon which other classes may be built.

Allows the base class to provide only an interface for its derived classes.

Prevents anyone from creating an instance of this class.

A class is made abstract if at least one pure virtual function defined.

How constructor and Destructor calls during Inheritance

A derived class inherits the members of its base class. Therefore, when a derived class object is instantiated, its base class members must be initialized in addition to its own members.

The base class constructor is called to initialize the base class members of the derived class object. The base class constructor, similar to component objects in a composite class, is called before the derived class object's constructor. Destructors are called in the reverse order.

Example:

```
class base { public:
base()
{
cout<< "Constructing base ... \n";
}
~base()
{
cout<< "Destroying base ... \n"; }
```

```
};
class derived : public base
{
public: derived()
{
cout<< "Constructing derived... \n";
}
~derived()
{
cout<< "Destroying derived... \n";
}
};
int main()
{
derived ob; return 0;
}
```

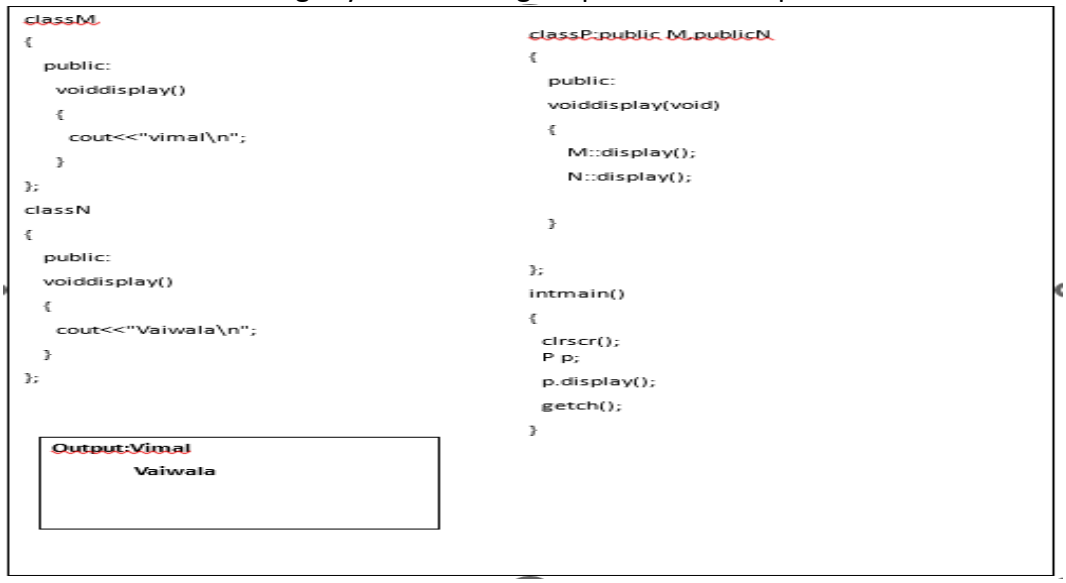
Output

```
Constructing base ...
Constructing derived...
Destroying derived...
Destroying base ...
```

Ambiguity of Multiple Inheritance

Ambiguity: - In multiple inheritance the ambiguity arises when same method name is being used by two derived class and further derivation from these two base classes.

→To resolve this ambiguity we are using scope resolution operator.



```
classM
{
  public:
    voiddisplay()
    {
      cout<<"vimal\n";
    }
};
classN
{
  public:
  voiddisplay()
  {
    cout<<"Vaiwala\n";
  }
};
```

```
Output:Vimal
       Vaiwala
```

```
classP:public M,publicN
{
  public:
  voiddisplay(void)
  {
    M::display();
    N::display();

  }

};
intmain()
{
  clrscr();
  P p;
  p.display();
  getch();
}
```