UNIT II
Operator Overloading & Inheritance

Developed By : Mrs. K. M. Sanghavi

Operator Overloading

Classes And Objects

OOP

- Introduction
- Overview of OOP
- Benefits of OOP
- Overview of C++
- Application of C++
- Conversion

Introduction

- What is overloading
  - Overloading means assigning multiple meanings to a function name or operator symbol
  - It allows multiple definitions of a function with the same name, but different signatures.
- C++ supports
  - Function overloading
  - Operator overloading

Operator Overloading

- "Operator overloading is the **ability** to **tell** the **compiler** how to perform a certain **operation** when its corresponding **operator** is used on one or more variables."

Function Overloading

```
void swap(int&, int&) //swap version– 1
void swap(Date&, Date&) //swap version– 2

main()
{
        int i, j;
        Date a, b;
        swap( i,  j );  //swap version – 1 applied
        swap( a, b ); //swap version – 2 applied
}
```

Overloaded Constructors


```
class Account
{       double  balance;
        double  rate;
public:
        Account();
        Account( double bal );
        Account( double bal, double pcnt);
        …
};
main()
{
        Account a1;
        Account a2(12000, 0.8);  }
```

Need Of Operator Overloading

- Operator overloading provides a convenient notation for manipulating user-defined objects with conventional operators.

Overloadable Operators

- arithmetic, logical, relational operators
- call (), subscript [], de-reference ->
- assignment and initialization
- explicit and implicit type conversion operators

Operator Overloading

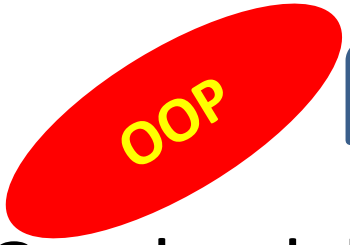
- Overloading of operators are achieved by creating **operator function**.

- "An ***operator function*** defines the operations that the overloaded operator can perform relative to the class".

- An operator function is created using the keyword **operator.**

Operator Function Definitons

- Two ways:
  - Implemented as member functions
  - Implemented as non-member or Friend functions
    - the operator function may need to be declared as a friend if it requires access to protected or private data



Oveloadable /Non-Overloadable Operators

Overloadable

+	-	*	/	%	^	&	|
~	!	=	<	>	<=	>=	+=
-=		*=	++	--	<<	>>	==
!=		&&	||	/=	%=	^=	&=
|=	*=	<<=	>>=	[ ]	( )	->	->*

new delete etc.

Non-overloadabale

:: .* . ?:

Operator Functions

- Unary ++, --, &, *, +, -, ~, !,
- Arithmetic +, -, *, /, %
- Shift <<, >>,
- Relational >, <, >=, <=, ==, !=
- Bitwise &, ^, |
- Logical &&, ||
- Assignment =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=,
- Data Access [], ->, ->*
- Function call()
- Comma ,

Operator Overloading

Creating a Member Operator Function

Within Class:

ret-type operator#(argument-list);

Outside Class:

ret-type class-name::operator#(arg-list)

{

// operations

}

Operator Function Definitons

1.Defined as a member function

```
class Complex {
   ...
 public:
   ...
   Complex operator +(Complex &op)
   {
         real   = real   + op._real,
         imag = imag + op._imag;
         return(Complex(real, imag));
   }
   ...
 };
```

Operator Overloading

Implicitly passed by ***this*** pointer

Explicitly passed **argument**

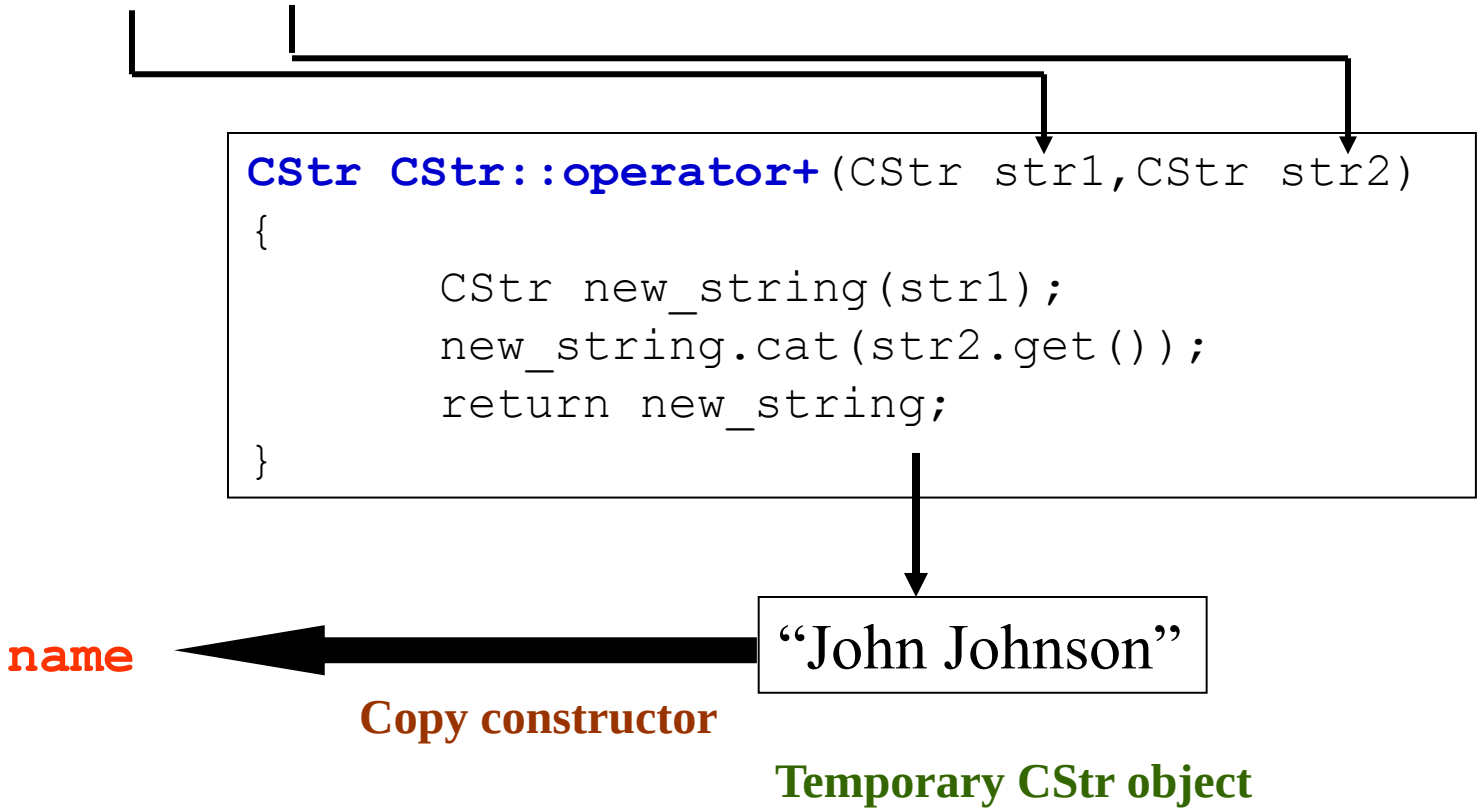
```
ob1 = ob1 + ob2;
```

```
ob1 = ob1.operator+ (ob2);
```

H does it work?

OOP

```
first("John");
CStr last("Johnson");
CStr name(first+last);
```



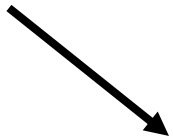
OOP

Operator Function Definitons

1.Defined as a friend function

```
class Complex {
    ...
  public:
    ...
   double real() { return _real; }
   double imag() { return _imag; }
friend Complex operator+(Complex, Complex)
  };
```

c = a+b;



c = operator+ (a, b);

```
Complex operator +(Complex &op1, Complex &op2)
{
        real   = op1.real   + op2.real,
             imag = op1.imag + op2.imag;
   return (Complex(real, imag));
}
```

Operator Functions as Class Members vs. as friend Functions

- Member vs non-member
  - –Operator functions can be member or non-member functions
  - –When overloading `( )`, `[ ]`, `->` or any of the assignment operators, must use a member function
- Operator functions as member functions
  - –Leftmost operand must be an object (or reference to an object) of the class
    - If left operand of a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - –Must be **`friend`**s if needs to access private or protected members
  - –Enable the operator to be commutative

OOP

Unary Operators

```
class UnaryExample
{
   private:
                int m_LocalInt;
            public:
                UnaryExample(int j)
                        {
                           m_LocalInt = j;
                        }
                        int operator++ ()
                        {
                           return (m_LocalInt++);
                        }
};
```

Unary Operators

```
void main()
{
    UnaryExample object1(10);
          cout << object1 ++; // overloaded operator
called
}
```

OOMP

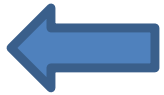
Binary Operators

```
class BinaryExample
{
   private:
              int m;
           public:
              BinaryExample(int j)
                       {
                          m = j;
                       }
                       int operator+ (BinaryExample& rhsObj)
                       {
                          m =  m + rhsObj.m;
             return m;                          }   };
```

OOMP

Binary Operators

```
void main()
{
    BinaryExample object1(10), object2(20);
        cout << object1 + object2; // overloaded
operator
}
```



OOMP

Rules for Operator Overloading

–Only **existing operators** can be overloaded. New operators cannot be created.

–The overloaded operator must have **at least one operand that is of user-defined type.**

–We **cannot change the basic meaning** of an operator.

–Overloaded operators follow the syntax rules of the original operators.

–Operator functions cannot have **default arguments**.

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators **&**, *****, **+** and **−** each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

OOMP

Rules for Operator Overloading

–The following operators that cannot be overloaded:

sizeof Size of operator

. Membership operator

.* Pointer-to-member operator

: : Scope resolution operator

? ; Conditional operator

OOPCGL

Rules for Operator Overloading

–The following operators can be over loaded with the use of member functions and not by the use of friend functions:

- Assignment operator =
- Function call operator( )
- Subscripting operator [ ]
- Class member access operator ->

–Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument.

OOMP

Rules for Operator Overloading

–Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

–When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

–Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

OOMP

Program for Implementation

•Pgm to create a class Matrix assuming its properties and add two matrices by overloading + operator. Read and display the matrices by overloading input(>>) & output(<<) operators respectively.

•Pgm to create a class RATIONAL with numerator and denominator as properties and perform following operations on rational numbers.

- **r** = **r1** * **r2**; (by overloading * operator)
- To check equality of **r1** and **r2** (by overloading == operator)

OOMP

Overloading =

- **Prototype:**

  classname& operator =(classname & obj);

- **Example:**

```
A & operator =(const A& obj)
{
   m = obj.m;
   return * this;
}
```

```
void main()
{
   A ob1( 10) ,ob2 ;
    :
   ob2 = ob1;
}
```

OOMP

Overloading >>

- **Prototype:**

  friend istream& operator >>(istream&, Matrix&);

- **Example:**

```
istream& operator >>(istream& in, Matrix& m)
{
   for(int i=0; i<row*col; i++)
   {
         in >> m[i];
    }
    return in;
}
```

```
void main()
{
  :
Cin>>mobj;
  :
}
```

OOMP

Overloading <<

- **Prototype:**

  friend ostream& operator <<(ostream&, Matrix&);

- **Example:**

```
ostream& operator <<(ostream& out, Matrix& m)
{
   for(int i=0; i<row; ++i)
{
    for(int j=0; j<col; j++)
    {  out>> m[i][j] >> "\t" ;  }
    out << endl;
 }
```

```
void main()
{
    :
cout<<mobj;
    :
}
```

Type Conversions

–The type conversions are automatic only when the data types involved are built-in types.

```
int m;
float x = 3.14159;
m = x; //  convert x to integer before its value is assigned
       //  to m.
```

–For user defined data types, the compiler does not support automatic type conversions.

–We must design the conversion routines by ourselves.

OOMP

Type Conversions

Different situations of data conversion between incompatible types.

- Conversion from basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type.

OOMP

Basic type to Class Type

```
class time
{     int hrs ;
      int mins ;
  public :
     …
  time (int t)
  {
      hrs = t / 60 ;
      mins = t % 60;
  }
} ;

void main ()
{    int duration = 85;
  time T1(duration) ;      // time t1 = duration; }
```

OOMP

Class type to Basic Type

A constructor function do not support type conversion from a class type to a basic type.

An overloaded ***casting operator*** is used to convert a class type data to a basic type.

It is also referred to as ***conversion function***.

```
operator typename( )
{
   …
   …  ( function statements )
   …
}
```

This function converts a ***class type*** data to ***typename***.

OOMP

Class type to Basic Type

```
vector : : operator double( )
{
   double sum = 0;
   for (int i=0; i < size ; i++)
      sum = sum + v[i] * v[i];
   return sqrt (sum);
}
```

This function converts a vector to the square root of the sum of squares of its components.

The casting operator function should satisfy the following conditions:

- **It must be a class member.**
- **It must not specify a return type.**
- **It must not have any arguments.**

OOMP

Class type to Basic Type

–Conversion functions are member functions and it is invoked with objects.

–Therefore the values used for conversion inside the function belong to the object that invoked the function.

–This means that the function does not need an argument.

```
vector v1;
double  d = v1;
```

OOMP

One Class type to Other Class Type

objX = objY ; // objects of different types

- **objX** is an object of class **X** and **objY** is an object of class **Y**.
- The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**.
- Conversion is takes place from **class Y** to **class X**.
- **Y** is known as ***source class***.
- **X** is known as ***destination class***.

OOMP

One Class type to Other Class Type

Conversion between objects of different classes can be carried out by either a constructor or a conversion function.

Choosing of constructor or the conversion function depends upon where we want the type-conversion function to be located in the source class or in the destination class.

Inheritance in C++

Developed by Ms. K.M. Sanghavi

Inheritance

- "Inheritance is the mechanism which provides the power of **reusability** and **extendibility**."
- "Inheritance is the process by which one **object can acquire the properties of another object**."
- "Inheritance is the process by which **new** classes called ***derived*** **classes** are created from **existing** classes called ***base*** **classes**."
- Allows the creation of **hierarchical** classifications.

Advantages of Inheritance

Sometimes we need to repeat the code or we need repeat the whole class properties. So Inheritance helps in various ways.

1.) It saves memory space.
2.) It saves time.
3.) It will remove frustration.
4.) It increases reliability of the code
5.) It saves the developing and testing efforts.

Need of Inheritance

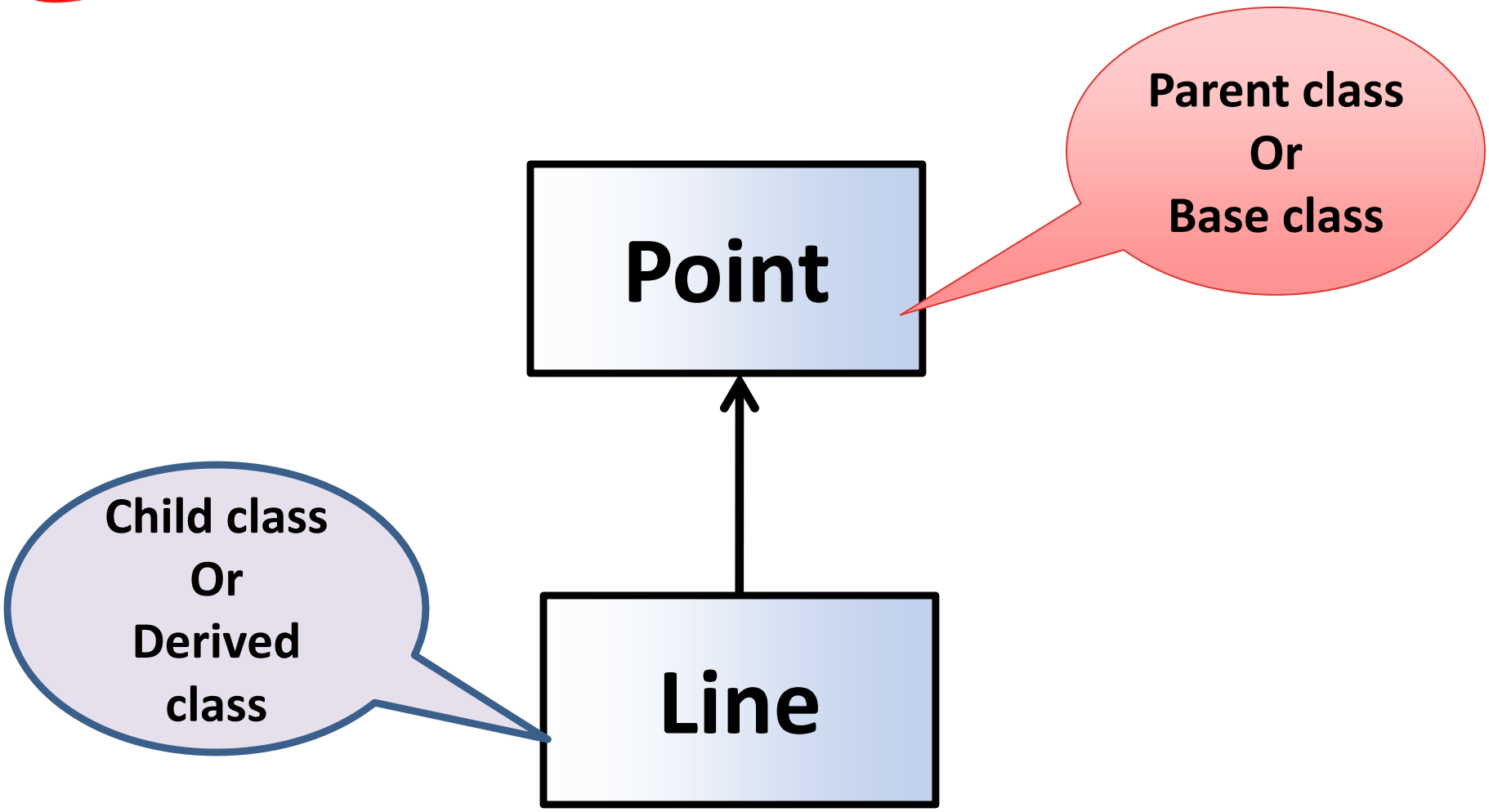
To increase the reusability of the code and to make further usable for another classes. We use the concept of inheritance.

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

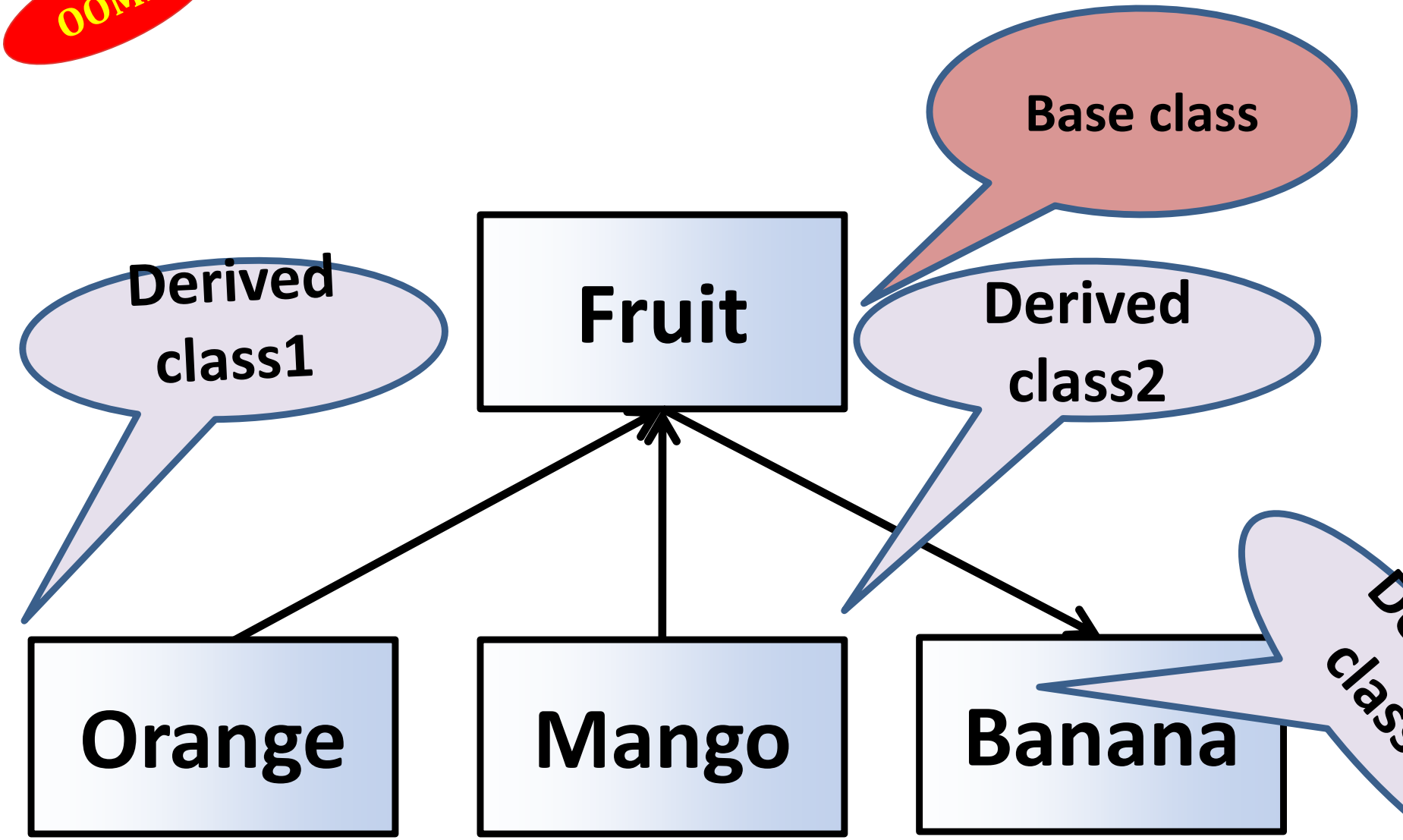
OOMP

Single Inheritance

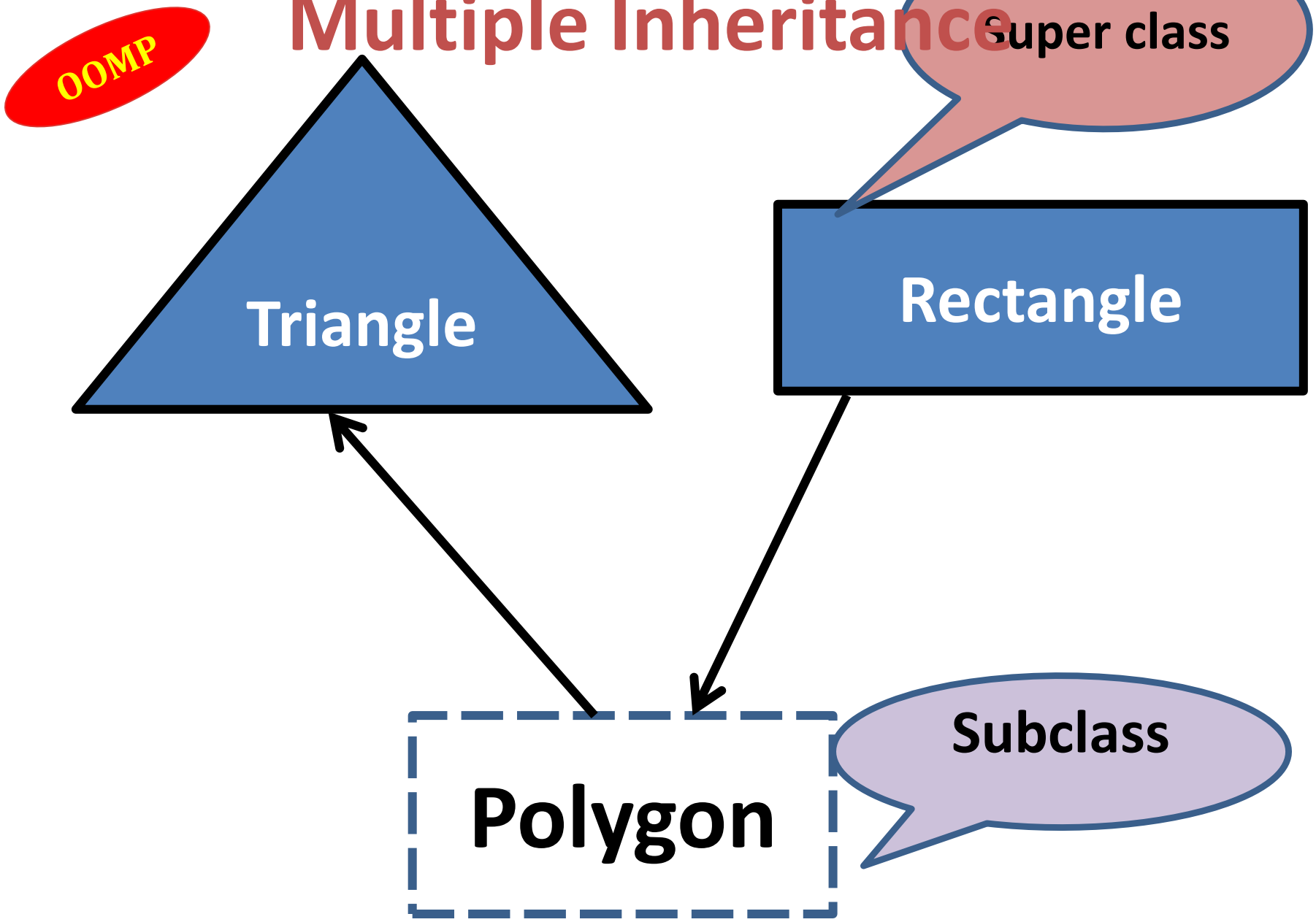


OOMP

Hierarchical Inheritance

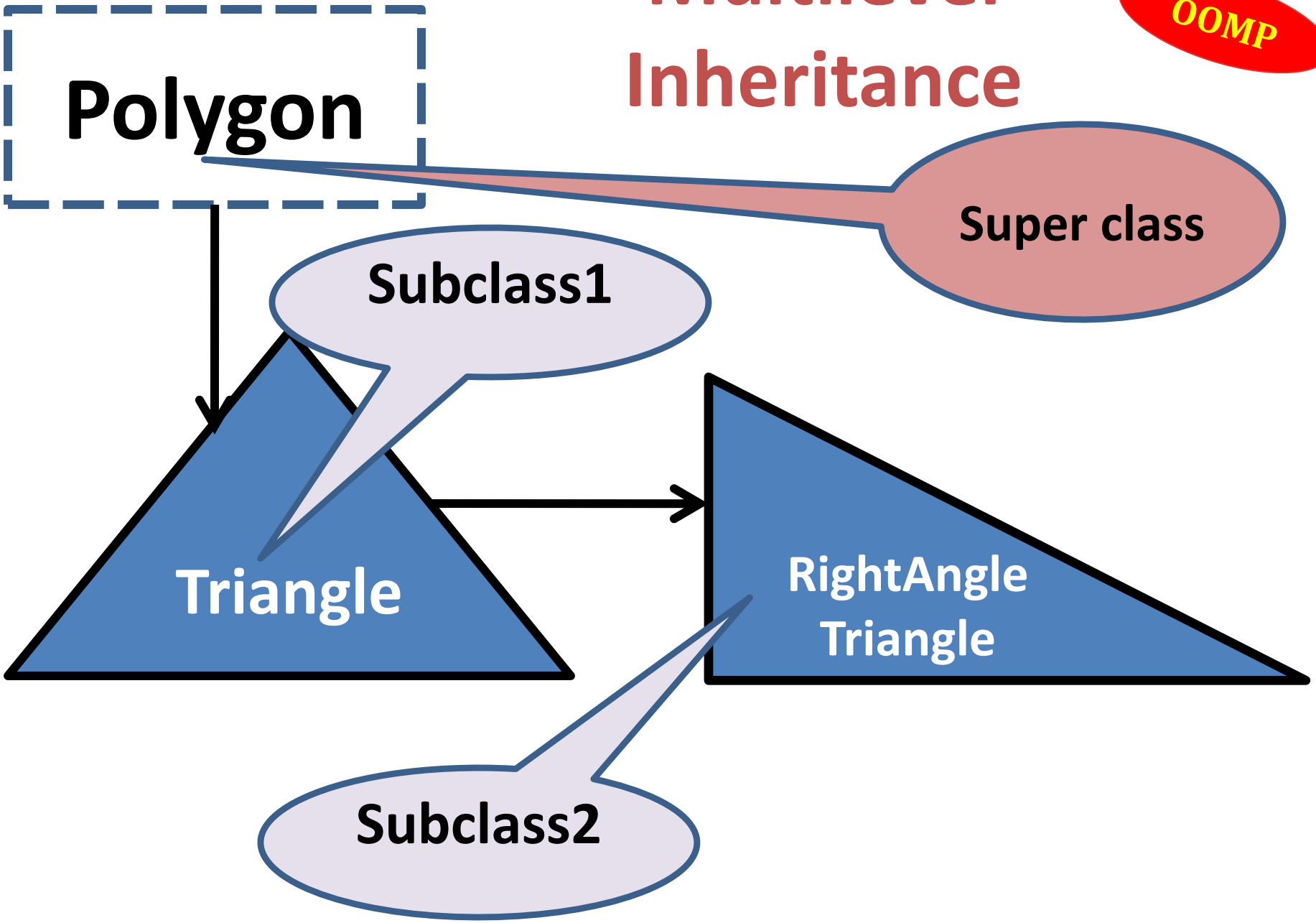


Multiple Inheritance

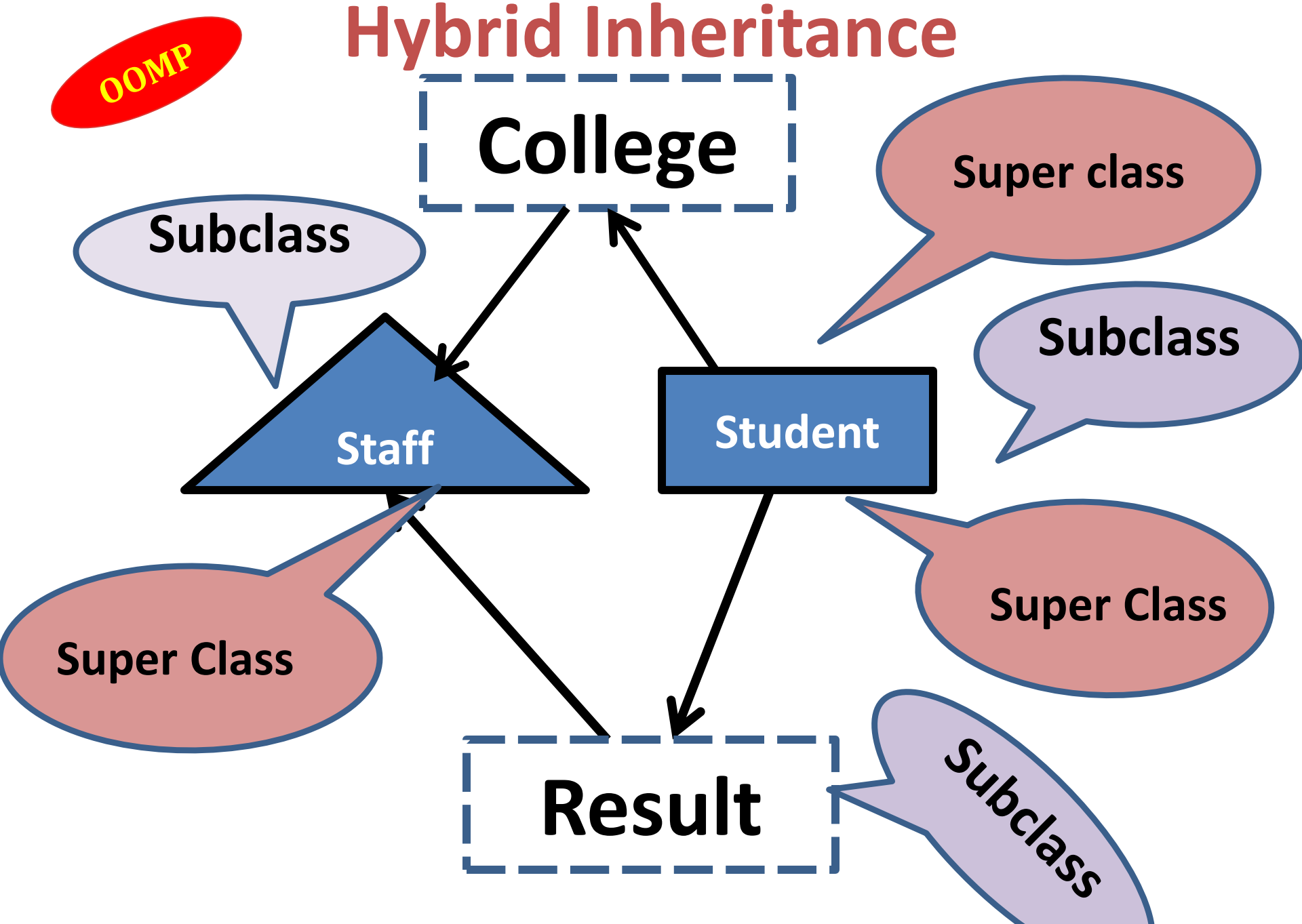


Multilevel Inheritance

OOMP



Hybrid Inheritance



Base Class

- **"Base class** is a class which defines those **qualities common to all objects** to be derived from the base."
- The base class represents the most **general description.**
- A class that is **inherited** is referred to as a base class.

Derived Class

- "The classes derived from the base class are usually referred to as **derived classes**."
- "A **derived class** includes **all** features of the generic base class and then adds **qualities specific** to the derived class."
- The class that does the **inheriting** is called the derived class.

Derived Class

Note:

Derived class can be used as a **base class** for another derived class.

- In C++, inheritance is achieved by allowing one class to incorporate another class into its declaration.

Access modes or visibility modes of inheritance

- We have seen two access modes in C++ classes: public and private
  - Public members are directly accessible by users of the class
  - Private members are NOT directly accessible by users of the class, not even by inheritors
- There is a 3rd access mode: protected
  - Protected members are directly accessible by derived classes but not by other users

Inheritance

- **Syntax:**

  class derived_class**: Acesss_specifier** base_class

  { };

- **Example:**

  class CRectangle**: public** Cpolygon{ };

  class CTriangle**: public** Cpolygon{ };

Access or Visibility mode of inheritance

OOMP

•Visibility of inherited base class members in derived class:-

Visibility mode is	Public Members of Base Class	PROTECTED MEMBERS OF BASE CLASS	Private Members of base class are not directly accessible to derived class.
Public Protected private	Public Protected private	Protected Protected private	

Inheritance

- **Syntax:**

```
class   baseclass
 {
     private int x;
     public int y;
    protected int z;
 }
```

OOMP

Inheritance : Private derivation

- **Syntax:**

```
class   derivedclass : private baseclass
 {
      x; // not accessible
      y;  // private
      z;  //protected
 }
```

OOMP

Inheritance : Public derivation

- **Syntax:**

```
class   derivedclass : public baseclass
 {
      x; // not accessible
      y;  // public
      z;  //protected
 }
```

Inheritance : Protected derivation

OOMP

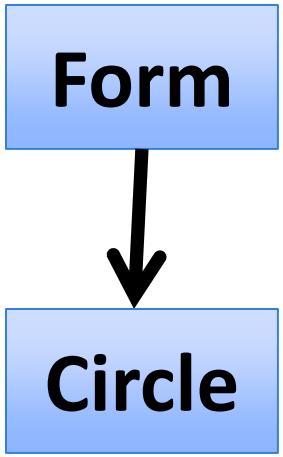
- **Syntax:**

```
class   derivedclass : protected baseclass
 {
      x; // not accessible
      y;  // protected
      z;  //protected
 }
```

Example for Single Inheritance

OOMP

```
class Form
 {
      private:
          double area;
      public:
         int color;
         double getArea()
             {
                 return this->area;
             }
          void setArea(double area)
             {
                 this->area = area;
             }
         bool isDark()
          {
            return (color > 10);
          }
};
```



Single Inheritance

```
class Circle : public Form
 {
      public:
           double getRatio()
           {
              double a;
              a = getArea();
              return sqrt(a / 2 * 3.14);
           }
           void setRatio(double radius)
            {  color = 5;
              setArea( pow(radius * 0.5, 2) * 3.14 );
           }
   };
```

Single Inheritance

```
void main( )
 {
     double ans;
     Circle c;
     c.setRatio(10);
     ans = c.getRatio( );
     cout<< ans;
     bool cAns = c.isDark( );
     cout<< cAns;
}
```

OOMP

Multiple Inheritance

- **Syntax:**

```
class   derivedclass : access-specifier baseclass1, access-specifier baseclass2
      {
                                        ……..
      }
```

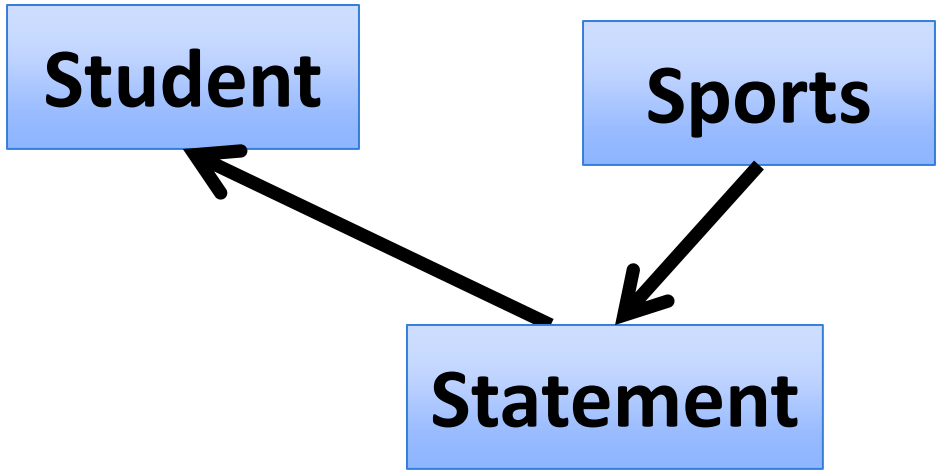
- **Example:**

```
class   result : public student, public staff
      {
                                        ……..
      }
```

OOMP

Example for Multiple Inheritance

```
#include<iostream.h>
#include<conio.h>
 class student
{
   protected:
     int rno,m1,m2;
   public:
            void get()
          {
                    cout<<"Enter the Roll no :";
                    cin>>rno;
                    cout<<"Enter the two marks   :";
                    cin>>m1>>m2;
          }
};
```



OOMP

Example for Multiple Inheritance

```
class sports
{
   protected:
     int sm;                  // sm = Sports mark
   public:
              void getsm()
            {
               cout<<"\nEnter the sports mark :";
               cin>>sm;

            }
};
```

OOMP

Example for Multiple Inheritance

```
class statement:public student,public sports
{
   int tot,avg;
   public:
   void display()
           {
             tot=(m1+m2+sm);
             avg=tot/3;
             cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal      : "<<tot;
            cout<<"\n\tAverage    : "<<avg;
           }
};
```

Example for Multiple Inheritance

```
void main()
{
   clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

OOMP

Hierarchical Inheritance

- **Syntax:**

```
class   derivedclass1 : access-specifier baseclass
      {
                                  ……..
      }

class   derivedclass2 : access-specifier baseclass
      {
                                  ……..
      }
```

- **Example:**

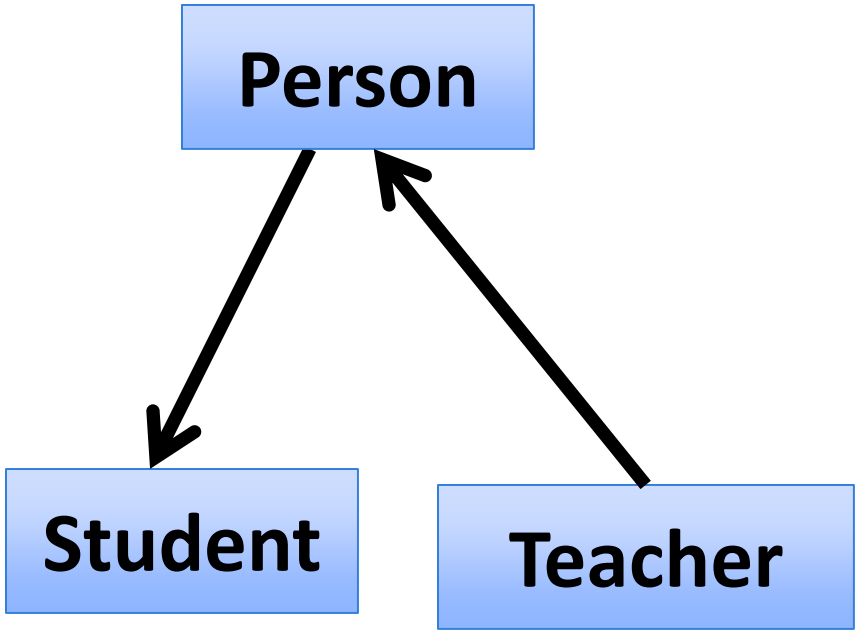
```
class  student : public person
      {
                        ……..
      }
```

```
class  staff : public person
{
                                  ……..
}
```

OOMP

Example for Hierarchical Inheritance

```
#include<iostream.h>
#include<conio.h>
class person
{
private:
char name[20];
long int phno;
public:
void read()
{
cout<<"\n Enter name ";
cin>>name;
cout<<"\n Enter Phno.=";
cin>>phno;
}
void show()
{
cout<<"\n Name="<<name;
cout<<"\n Phone="<<phno;
}
};
```



Example for Hierarchical Inheritance

```
class student : public person
{
private:
int rollno;
char course[20];
public:
void read()
{
person::read();
cout<<"\n Enter Roll No.=";
cin>>rollno;
cout<<"\n Enter Course=";
cin>>course;
}
void show()
{
person ::show();
cout<<"\n Roll No.="<<rollno;
cout<<"\n Course="<<course;
}
};
```

Example for Hierarchical Inheritance

```
class teacher : public person
{
private:
char dept_name[10];
char qual[10];
public:
void read()
{
person::read();
cout<<"\n Enter dept_name andQualification=";
cin>>dept_name>>qual;
}
void show()
{
person::show();
cout<<"\n Departement="<<dept_name;
cout<<"\n Qualififcation="<<qual;
}
};
```

Example for Hierarchical Inheritance

```
main()
{
clrscr();
student s1;
cout<<"\n**************** Enter student Information******************";
s1.read();
cout<<"\n*************** Displaying Student Information*************";
s1.show();
teacher t1;
cout<<"\n***************** Enter Teacher Information*****************";
t1.read();
cout<<"\n**************Displaying Teacher Information************";
t1.show();

getch();
}
```

OOMP

Multilevel Inheritance

- **Syntax:**

```
class   derivedclass1 : access-specifier baseclass1
      {
                                  ……..
      }

class   derivedclass2 : access-specifier derivedclass1
      {
                                  ……..
      }
```

- **Example:**

```
class   result : public staff
      {
                        ……..
      }
```

```
class salary : public staff
{
                                  ……..
}
```

OOMP

Inheritance and Constructors and Destructors

When an object of derived class is created , first the base class constructor is called , followed by the derived class constructor. When an object of a derived class expires , first the derived class destructor is invoked , followed by the base class destructor.

For example:

```
Class x{
                    };
Class y:public x
                {
                        };
```

OOMP

Answer

Constructor x
Constructor y
Destructor y
Destructor x

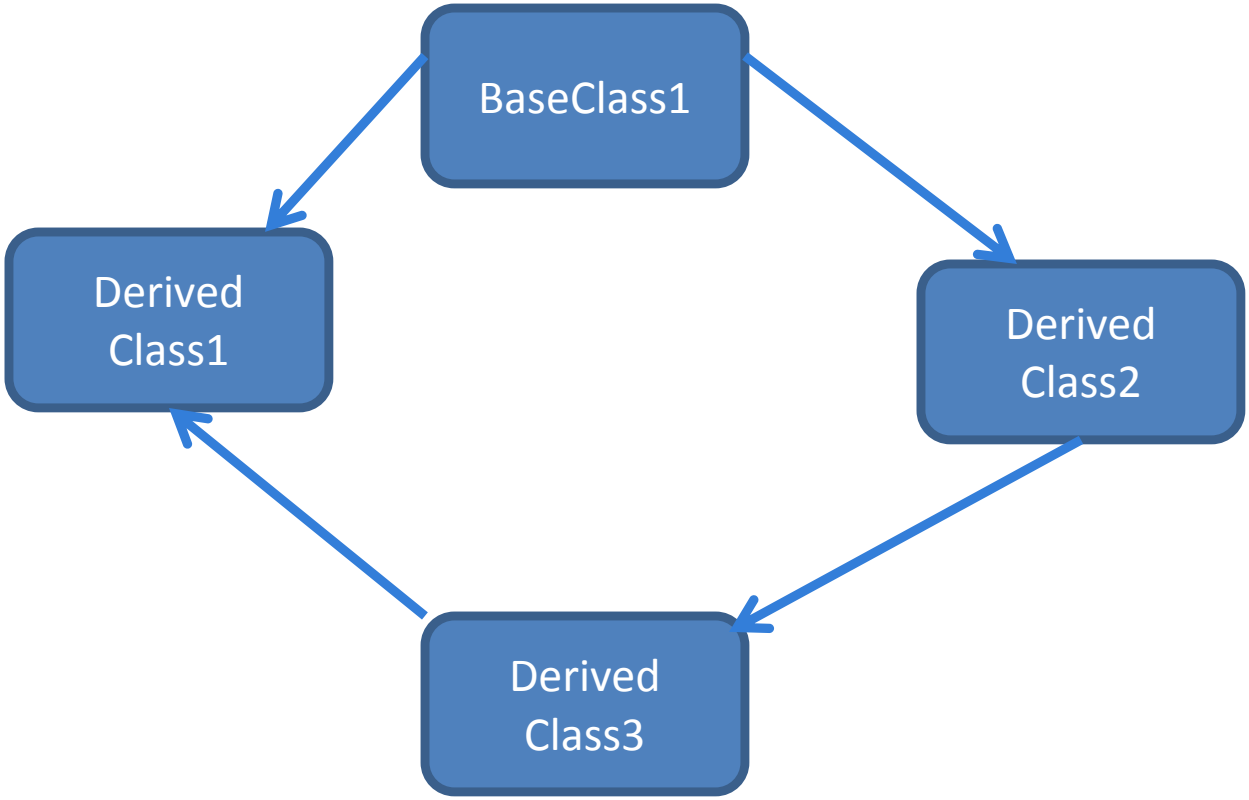
Passing Arguments to the Base class constructor

```
Class Base
{           int a;
            float b;
            public:
                        Base (int i , float j)
                        {
                                    a=i;
                                    b=j;
                        }
};
Class Derived : public Base
{
                        public:
                          Derived ( int p , float q) : Base (p , q)
                                    {                       }
};
```

Here even derived constructor does not need a parameter for itself , yet it accepts parameters for base constructor and then invokes base class constructor with these parameters.

OOMP

VIRTUAL BASE CLASS



VIRTUAL BASE CLASS

```
class baseclass1
{
   public:
     int a;
   public:
      baseclass1()
            {
               cout<<“baseclass1\n”;
             }
};
```

OOMP

VIRTUAL BASE CLASS

```
class derivedclass1 : public baseclass1
{
   public:
     int a1;
   public:
            derivedclass1()
             {
                cout<<“derived1\n”;
            }
};
```

```
class derivedclass2 : public baseclass1
{
   public:
     int a2;
   public:
            derivedclass2()
            {
                cout<<“derived2\n”;
            }
};
```

OOMP

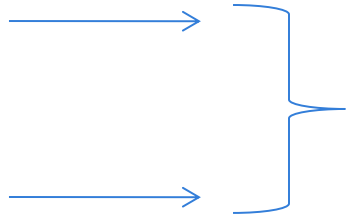
VIRTUAL BASE CLASS

```
class derivedclass3 : public derivedclass1,
derivedclass2
{
   private:
      int a3;
    public:
          derivedclass3()
          {
              a3 = 6;
             cout<<“derived3\n”;
           }
};

int main ( )
```

OUTPUT :

baseclass1
derived1
baseclass1
derived2
derived3



Two copies of baseclass1 in object of derivedclass3

OOMP

VIRTUAL BASE CLASS

```
int main ( )
 {
    derivedclass3 d3;
     cout<<d3.a;          //error : request for member ‘a’ is ambiguous
      retutn 0;
}
```

- Ambiguous because variable ‘a’ is **present twice in object of derivedclass3**, one in **derivedclass1** and other in **derivedclass2**

- **Because of two copies of ‘a’ present in object ’d3’ , the compiler does not know which copy is being referred.**

OOMP

VIRTUAL BASE CLASS

Remedy…..?

MANUAL SELECTION : Scope resolution operator

virtual base class

MANUAL SELECTION

This is done by using Scope resolution Operator:

```
int main ( )
 {
   derivedclass3 d3;

    d3.derivedclass1:: a = 10;

     d3.derivedclass2:: a = 20;

    cout<< d3.derivedclass1:: a<<"\n";
    cout<< d3.derivedclass2:: a<<"\n";
     return 0;
}
```

OUTPUT :

baseclass1
derived1
baseclass1
derived2
derived3

10
20

- **d3.derivedclass1** :: a and d3.**derivedclass2::a**

- In the above statements , use of scope resolution operator resolves the problem of ambiguity
- But it is not efficient way, How to make only one copy of ‘a’ available in object ‘d3’ which consumes less memory and easy to access without :: operator

- For this use VIRTUAL BASE CLASS

OOMP

Virtual Base Classes

- Used to **prevent** multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual when it is inherited.**

- **Syntax:**

```
class derived : virtual public base
{   .  .  .   };
```

OOMP

EXAMPLE : VIRTUAL BASE CLASS

```
class baseclass1
{
   public:
     int a;
   public:
      baseclass1()
            {
               cout<<“baseclass1\n”;
             }
};
```

OOMP

EXAMPLE :VIRTUAL BASE CLASS

```
class derivedclass1 : virtual public baseclass
{
   public:
     int a1;
   public:
            derivedclass1()
             {
                cout<<“derived1\n”;
            }
};
```

```
class derivedclass2 : virtual public baseclass
{
   public:
     int a2;
   public:
            derivedclass2()
            {
               cout<<“derived2\n”;
            }
};
```

OOMP

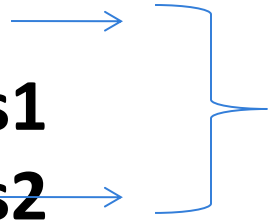
EXAMPLE :VIRTUAL BASE CLASS

```
class derivedclass3 : public derivedclass1,
derivedclass2
{
   private:
      int a3;
    public:
          derivedclass3()
          {
              a3 = 6;
             cout<<“derived2\n”;
           }
};

int main ( )
```

OUTPUT :

Baseclass1
derivedclass1
derivedclass2
derivedclass3

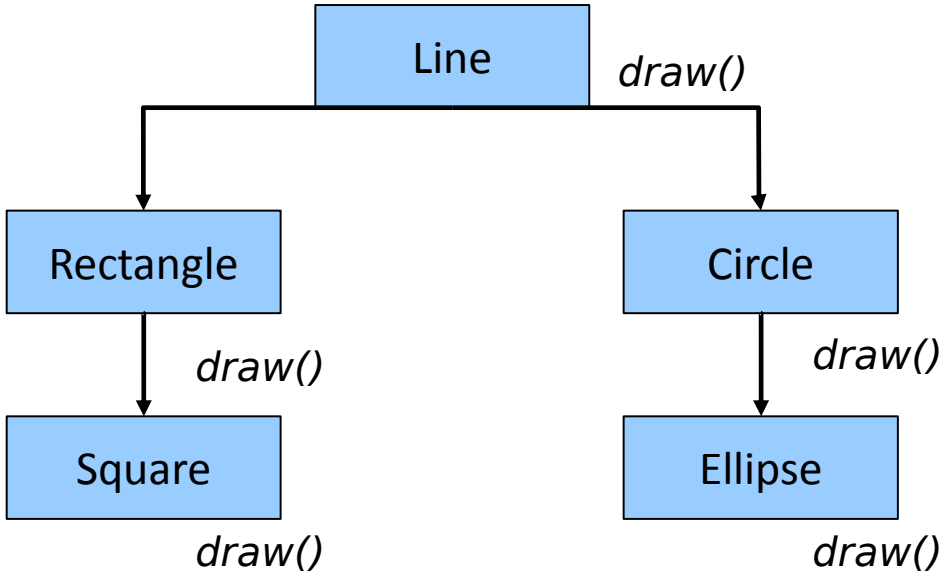


Only one copy maintained , no ambiguity

OOMP

Motivation: Virtual Functions

- a hierarchy of geometric shape classes
  - draws circles, ellipses, rectangles etc
- just use method **draw** throughout the hierarchy



Pointers to Derived Classes

- C++ allows base class pointers to point to derived class objects.
- Let we have
  - class base { … };
  - class derived : public base { … };
- Then we can write:

```
base *p1;
derived d_obj;
p1 = &d_obj;
```

```
class Base {
public:
  void show() {
    cout << “base\n”;
  }
};
class Derv1 : public base {
public:
  void show() {
    cout << “derived1\n”;
  }
class Derv2 : public base {
public:
  void show() {
    cout << “derived2\n”;
  }
};
```

```
void main()
{

Derv1 dv1;
Derv2 dv2;

Base *ptr
ptr = &dv1;
ptr->show();

 ptr = &dv2;
 ptr ->show();

return 0;
}
```

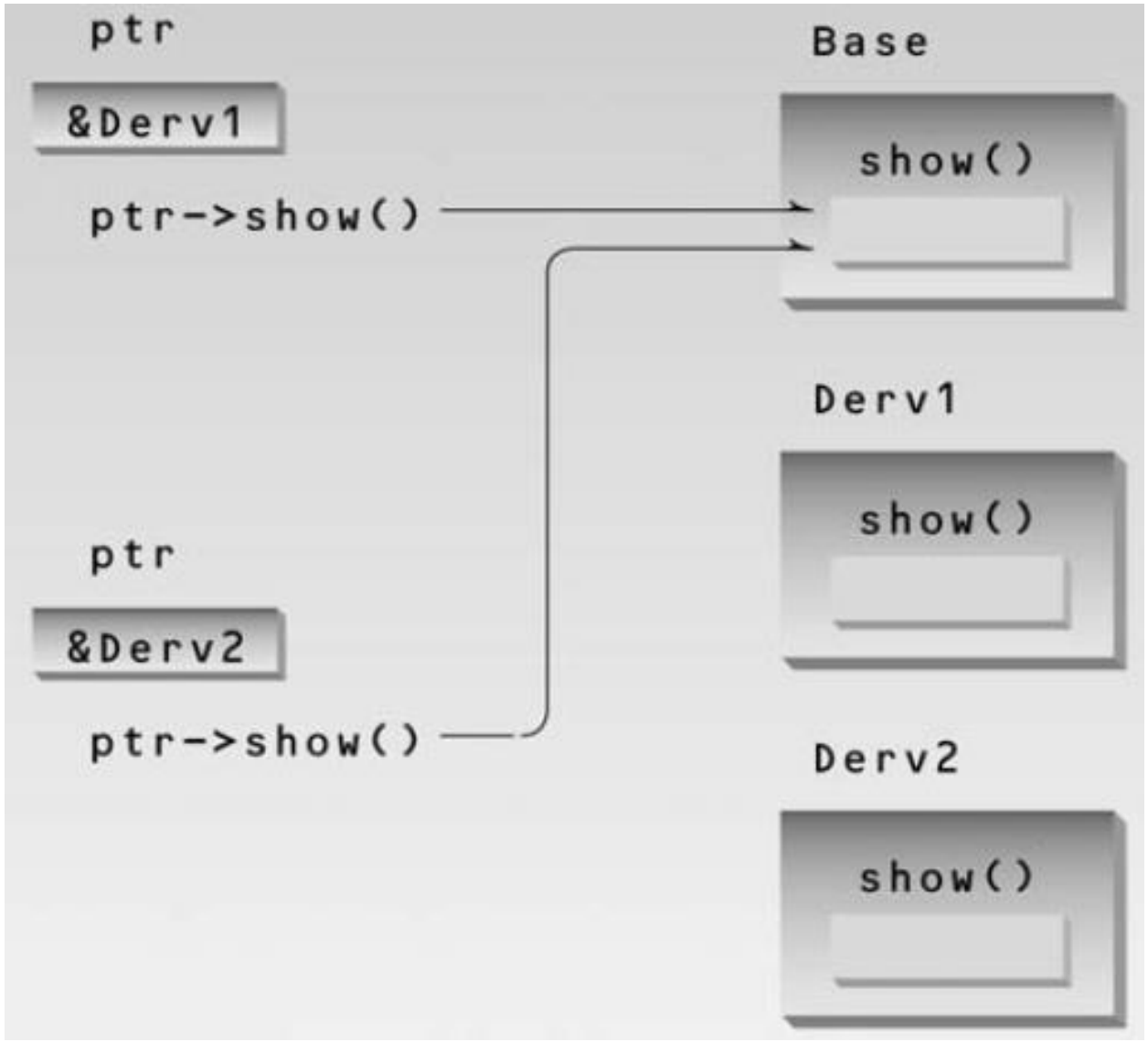
OUTPUT :

base

base

OOMP

Non-Virtual Pointer Access



```
class Base {
public:
  virtual void show() {
     cout << “base\n”;
   }
};
class Derv1 : public base {
public:
   void show() {
     cout << “derived1\n”;
   }
class Derv2 : public base {
public:
   void show() {
     cout << “derived2\n”;
   }
};
```

```
void main()
{

Derv1 dv1;
Derv2 dv2;

Base *ptr
ptr = &dv1;
ptr->show();

 ptr = &dv2;
 ptr ->show();

return 0;
}
```

OUTPUT :

derived1

derived2

OOMP

Virtual Functions

- "A virtual function is a member function that is declared **within** a **base class** and **redefined** by a **derived class**."

- Virtual functions implements the **"one interface, multiple methods"** philosophy under polymorphism.

OOMP

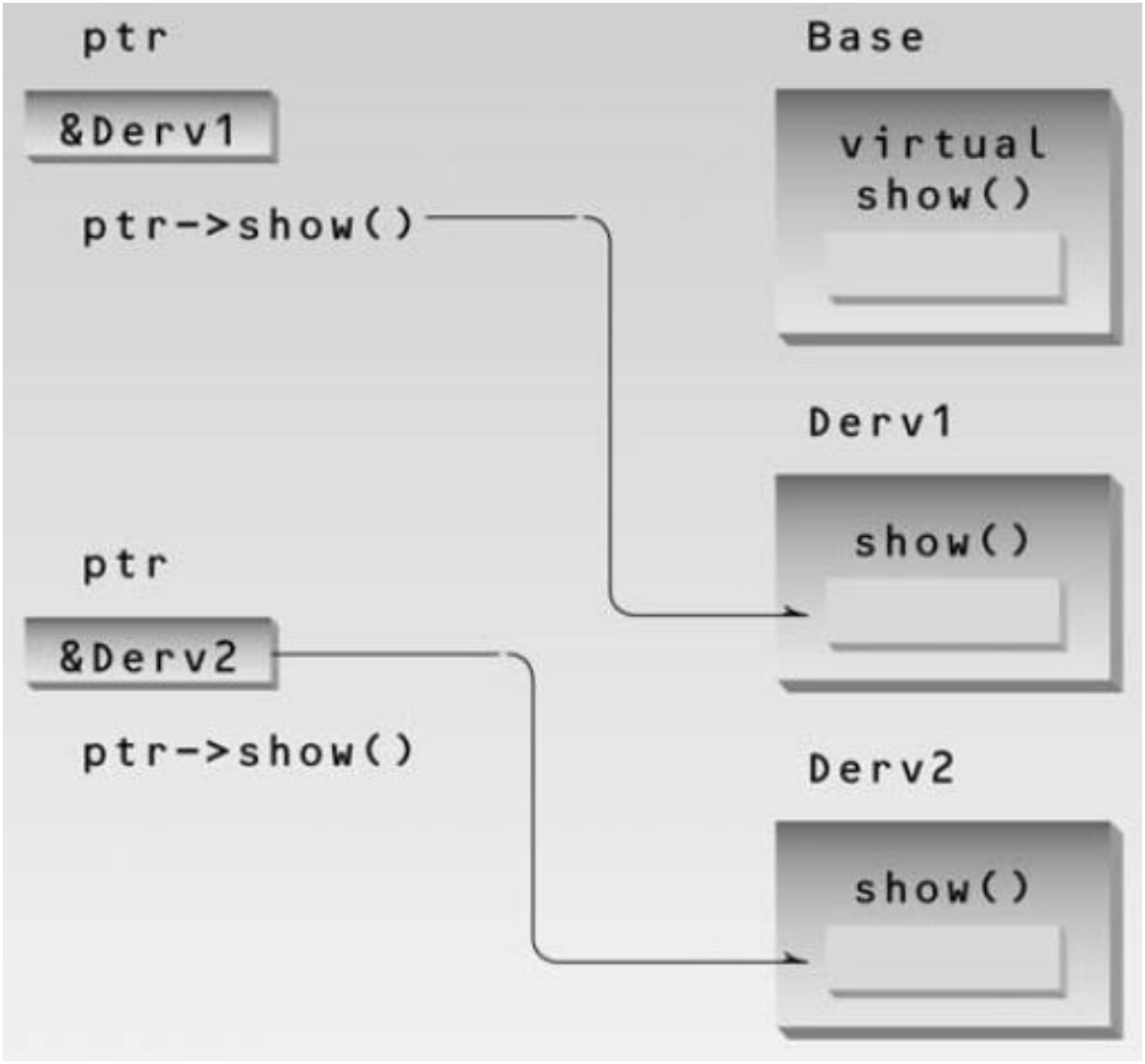
Virtual Functions

- The virtual function within the base class defines the form of the **interface** to that function.

- Each **redefinition** of the virtual function by a derived class implements **its operation** as it relates specifically to the derived class. That is, the redefinition creates a **specific** method.

OOMP

Non-Virtual Pointer Access



OOMP

Virtual Functions

- "A virtual function is a member function that is declared **within** a **base class** and **redefined** by a **derived class**."

- Virtual functions implements the **"one interface, multiple methods"** philosophy under polymorphism.

OOMP

Virtual Functions

- The virtual function within the base class defines the form of the **interface** to that function.

- Each **redefinition** of the virtual function by a derived class implements **its operation** as it relates specifically to the derived class. That is, the redefinition creates a **specific** method.

OOMP

Virtual Functions

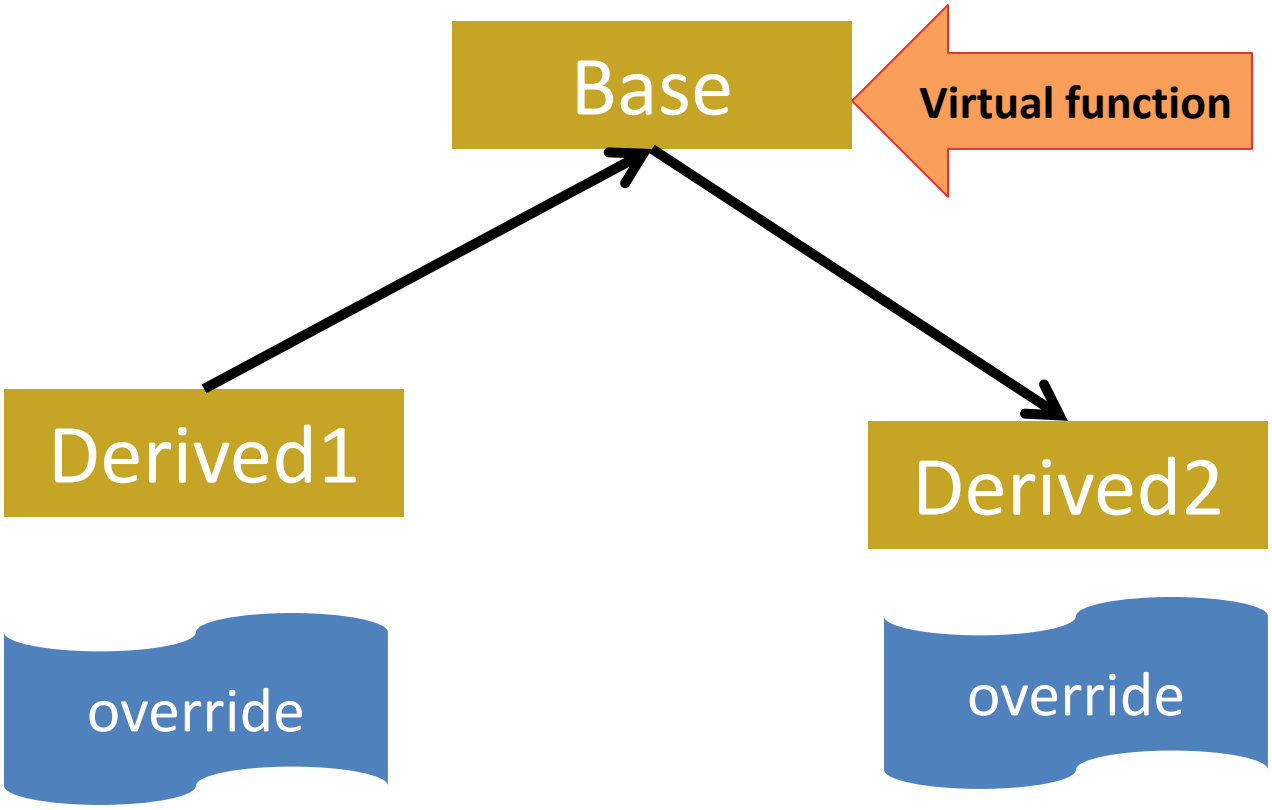
- To create a virtual function, precede the function‘s declaration in the base class with the keyword **virtual.**

- **Example:**

```
class base {
    public:
        virtual void member_func(){   }
};
```

OOMP

Virtual Functions



OOMP

Virtual Functions

- **How to implement run-time polymorphism?**
  - create base-class pointer can be used to point to an object of any class derived from that base
  - initialize derived object(s) to base class object.

- Based upon which derived class objects' assignment to the base class pointer, c++ determines which version of the virtual function

OOMP

Virtual Functions

- The **redefinition of a virtual function** by a derived class appears similar to **function overloading**?
- No

- The prototype for a redefined virtual function must match exactly the prototype specified in the base class.

OOMP

Virtual Functions

Restrictions:

- All aspects of its prototype must be the **same** as base class virtual function.
- Virtual functions are of **non-static** members.
- Virtual functions can not be **friends**.
- **Constructor** functions can**not** be **virtual**.
- But **destructor** functions **can** be **virtual**.

NOTE:

Function overriding is used to describe virtual function redefinition by a derived class.

OOMP

Virtual Functions

- When accessed **"normally"** virtual functions behave just like any other type of class member function.

- But virtual functions' importance and capacity lies in supporting the **run-time polymorphism** when they accessed via a pointer.

OOMP

Abstract Classes and Pure Virtual Functions

- A base class whose objects are never instantiated is called an abstract class. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects.
- A base class made as an abstract class must contain at least one pure virtual function.
- A pure virtual function is one with the expression "=0" added to the function declaration.

e.g.,

```
virtual void show() = 0;
```

OOMP

Pure Virtual Functions

- When a **virtual function** is made **pure**, any **derived** class must **provide** its **definition**.
- If the **derived** class **fails** to override the pure virtual function, a **compile-time error** will result.

NOTE:
When a virtual function is declared as pure, then all derived classes must override it.

```
class Base {
public:
  virtual void show()  = 0;
};

class Derv1 : public base {
public:
   void show() {
      cout << “derived1\n”;
   }

class Derv2 : public base {
public:
   void show() {
      cout << “derived2\n”;
   }
};
```

```
void main()
{

Base bad;        //ERROR

Derv1 dv1;
Derv2 dv2;

Base *ptr
ptr = &dv1;
ptr->show();

 ptr = &dv2;
 ptr ->show();

return 0;
}
```

Virtual Destructors – Without

- Normally, when one deletes an instance of a derived class (e.g., `Car`), the destructors of the derived class and those of all the ancestor classes are executed (in this case, the `Vehicle` destructor)

- But let's assume that we are given the following statement

```
Vehicle* a = new Car("Ferrari");
```

- What happens when one executes the following statement?

OOMP

Virtual Destructors – Why?

- Since the classes involved in the example do not have virtual destructors, only the `Vehicle` destructor is executed!
- Further, if additional classes appeared in the hierarchy between `Vehicle` and `Car`, their destructors would not be executed, either
- This behavior can lead to memory leaks and other problems, especially when dynamic memory or class variables are managed by the derived class
- A solution to the problem is the use of *virtual*

Virtual Destructors – What?

- A virtual destructor is simply a destructor that is declared as a virtual function
- If the destructor of a base class is declared as virtual, then the destructors of all its descendant classes become virtual, too (even though they do not have the same names)

OOMP

Virtual Destructors - rule

- Rules of thumb for virtual destructors
  - –If any class in a hierarchy manages class variables or dynamic memory, make its destructor virtual
  - –If none of the classes in a hierarchy have user-defined destructors, do not use virtual destructors

```
#include iostream.h


class Base
{
public: Base()
{
   cout<<"Constructing Base";

}

 // this is a destructor:
    ~Base(){ cout<<"Destroying Base";}

};
```


```
class Derive: public Base
 {
public: Derive()
{
cout<<"Constructing Derive";
}

~Derive()
  {
      cout<<"Destroying Derive";
}
 };

void main()

   {

            Base *basePtr = new Derive();
            delete basePtr;

   }
```

OUTPUT :

Constructing Base
Constructing Derive
Destroying Base

- Based on the output above, we can see that the constructors get called in the appropriate order when we create the Derive class object pointer in the main function.
- But there is a major problem with the code above: the destructor for the "Derive" class does not get called at all when we delete ‘basePtr’

OOMP

Remedy…..?

Well, what we can do is make the base class destructor virtual, and that will ensure that the destructor for any class that derives from Base (in our case, its the "Derive" class) will be called.


```
#include iostream.h


class Base
{
public: Base()
{
   cout<<"Constructing Base";

}

 // this is a destructor:
 virtual   ~Base(){ cout<<"Destroying Base";}

};
```


```
class Derive: public Base
 {
public: Derive()
{
cout<<"Constructing Derive";
}

~Derive()
  {
      cout<<"Destroying Derive";
}
 };

void main()

   {

            Base *basePtr = new Derive();
            delete basePtr;

   }
```

OUTPUT :

Constructing Base
Constructing Derive
Destroying Derive
Destroying Base

OOMP

Early vs. Late Binding

- "Early binding refers to events that occur at **compile time**."
- Early binding occurs when all information needed to call a function is known at compile time.
- **Examples :**

  function calls ,overloaded function calls, and overloaded operators.

Early vs. Late Binding

- "Late binding refers to function calls that are **not resolved until run time**."
- Late binding can make for somewhat **slower** execution times.
- **Example:**

  virtual functions

Containership

- If we create the object of one class into another class and that object will be a member of the class , then it is called containership.
- This relation is calles has_a relation.
- While the inheritance is called kind_of and is_a relation.


```
Class upper
{
Public:
Void display()
{
Cout<<“hello”<<endl;
}
};
Class  lower
{
Upper obj;
Public:
Lower()
{
Obj.display();
}

};
```

```
Void main()
{
Lower l;
}
```

o/p

hello

OOMP

Container class also call upper class constructor first like inheritance.It can use only public member of upper class not the private and protected members

OOMP

```
Class upper
{
Public:
upper()
{
Cout<<“it is upper class constructor
<<endl”;
}

};
Class lower
{
Upper obj;
Public:
lower()
{
Cout<<“it is lower class constructor
<<endl”;
}
};
```

```
Void main()
{
lower l;
}
```

o/p

it is upper class constructor
it is lower class constructor