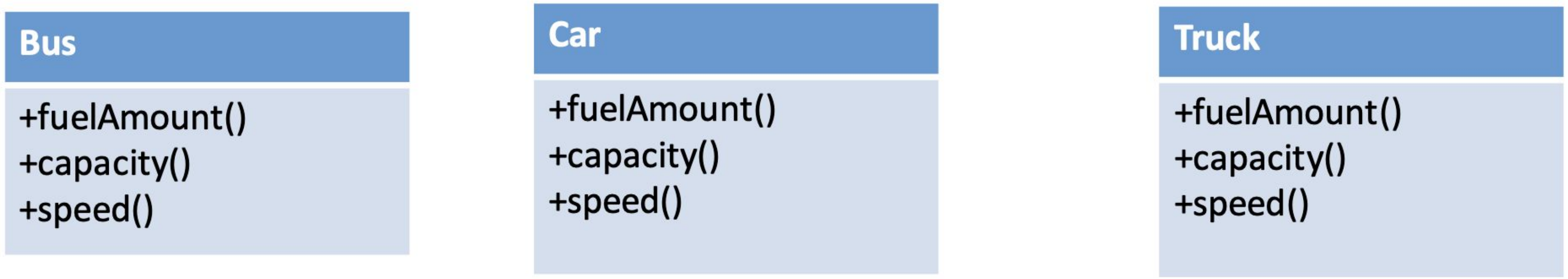
Object Inheritance and Reusability

Chapter 4

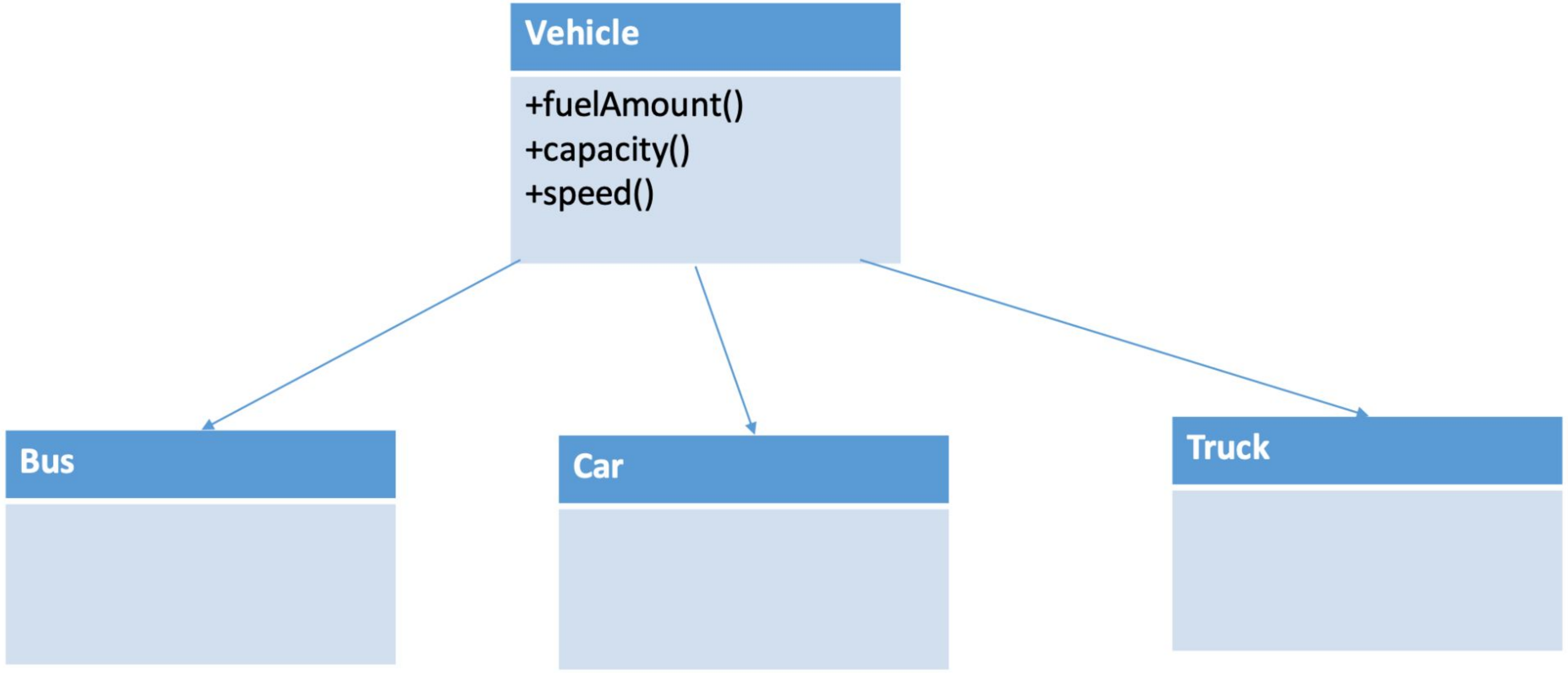
Inheritance

- Inheritance allows us to create new class from already existing class
- Inheritance is a process where the child class acquires properties and functionality of its parent class
- The Class that inherits another class is referred as child class, derived class or subclass
- The already existing class through which new class is inherited is referred as parent class, base class or super class

Without using inheritance:



Using inheritance:



Inheritance

Syntax for defining a derived class is :

```
class baseClassName {

        …   //code for base class

        };

class derivedClassName : visibility_mode baseClassName {

         …   //code specific for derived class

        };
```

Example:

```
class ABC {
        code for base class ABC
        };
class XYZ : public ABC {
code specific for derived class XYZ
};
```

Visibility modes / Inheritance modes

Visibility modes specifies whether the features of the base class is derived in private, public or protected mode

Types:

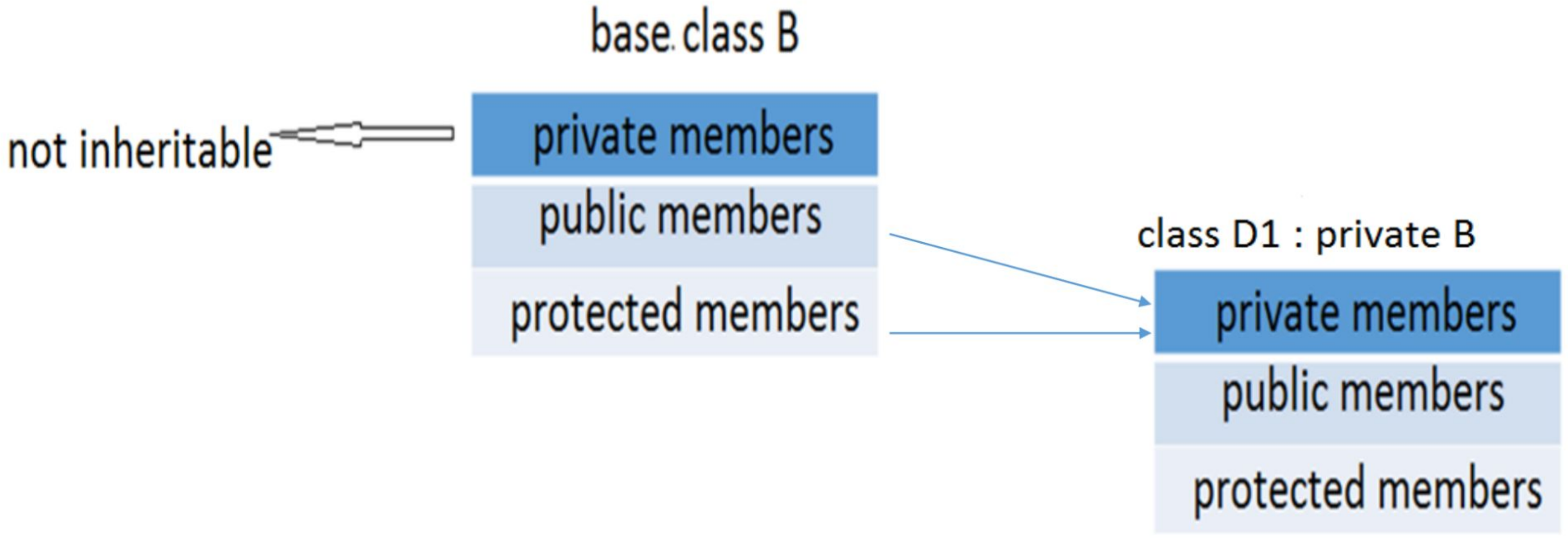
1. Private mode
2. Public mode
3. Protected mode

1. Private mode

• When the base class is inherited by derived class in private mode, both the public member and protected members of base class will become private member in derived class.

Note: The private members of base class are not directly accessible in derived class however, they can be accessed using public or protected methods of base class in the derived class.

1. Private mode:

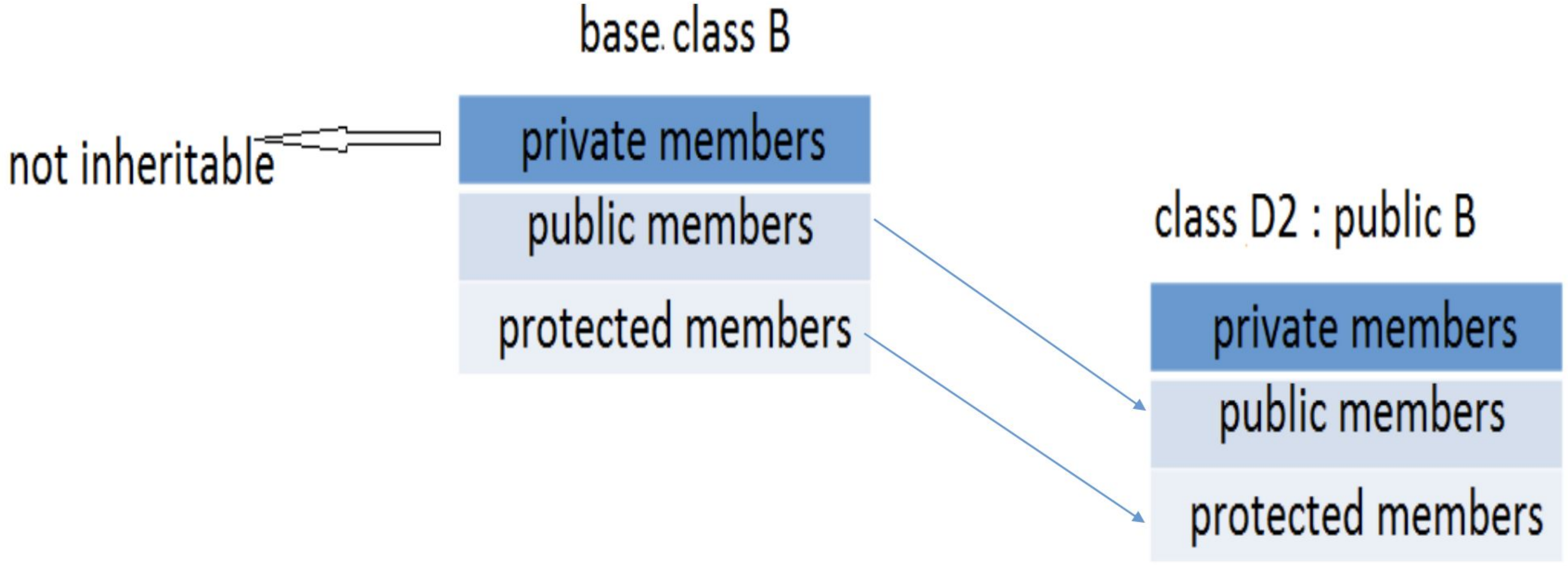


2. Public mode

• When the base class is inherited by derived class in public mode, public members of base class will become public members and protected members will become protected members for derived class

Note: The private members of base class are not directly accessible in derived class however, they can be accessed using public or protected methods of base class in the derived class.

2. Public mode:



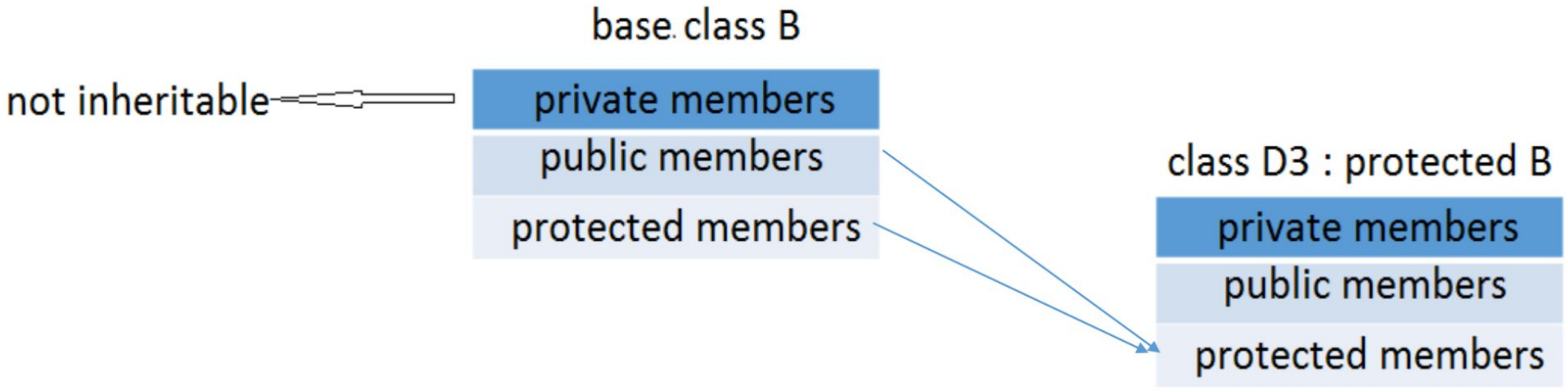
3. Protected mode

• When the base class is inherited by derived class in protected mode, public members and protected members of base class will become protected members for derived class

Note: The private members of base class are not directly accessible in derived class however, they can be accessed using public or protected methods of base class in the derived class.

Note: Protected members can be accessed within same class and its derived classes only

3. Protected mode:



Types of inheritance

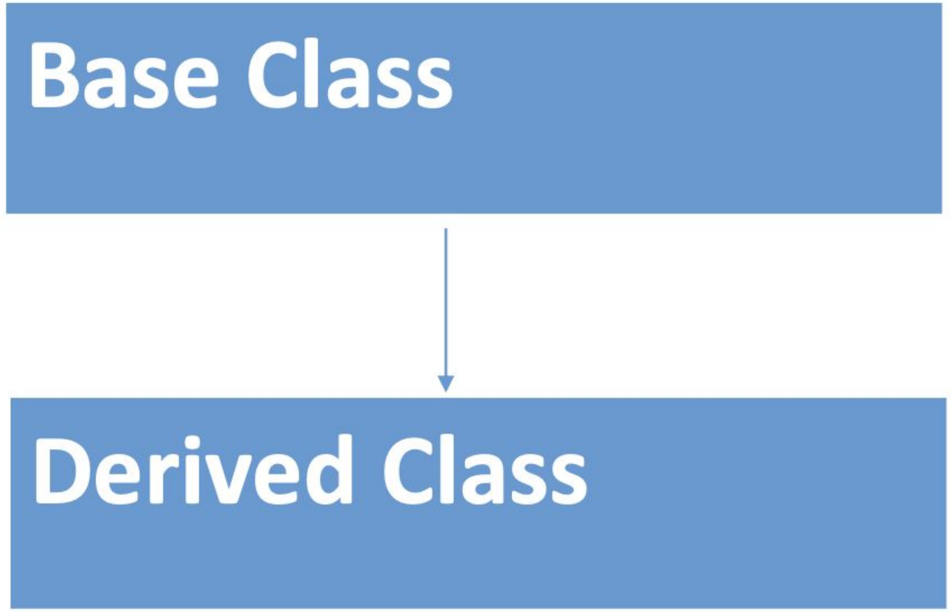
1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

1. Single Inheritance:

• When a single child class is being inherited by a single parent class, it is called single inheritance

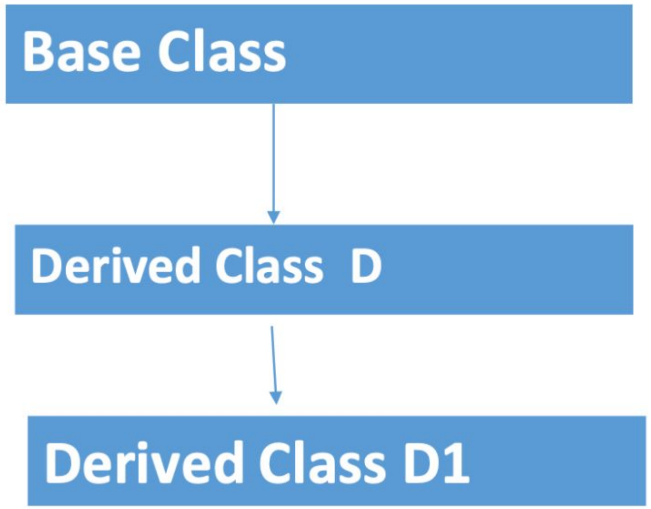
• i.e. one derived class with only one base class

1. Single Inheritance:



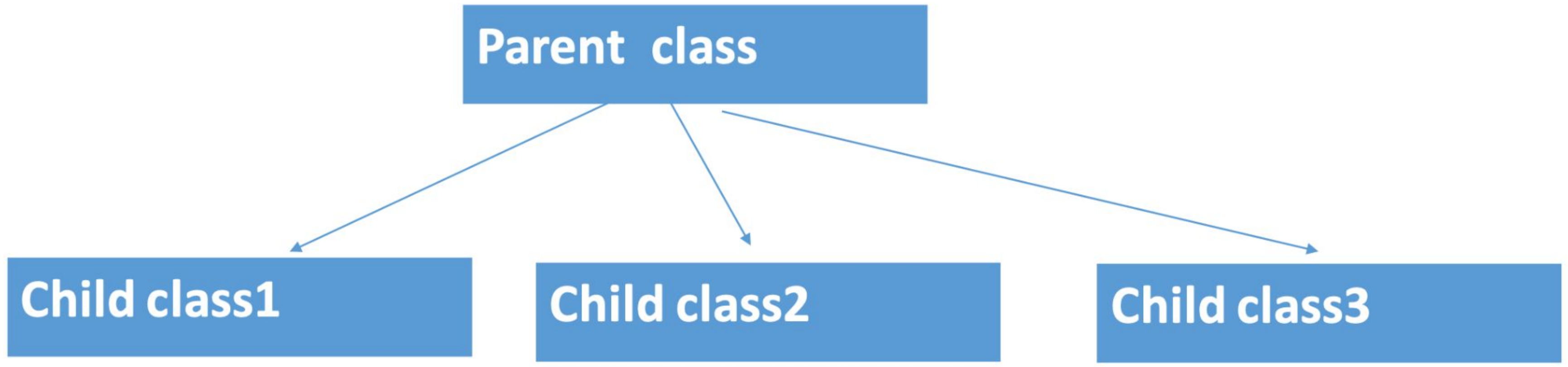
2. Multilevel Inheritance:

- In multilevel inheritance, a class is derived from another derived class.
- The base class of a derived class is derived class of another base class



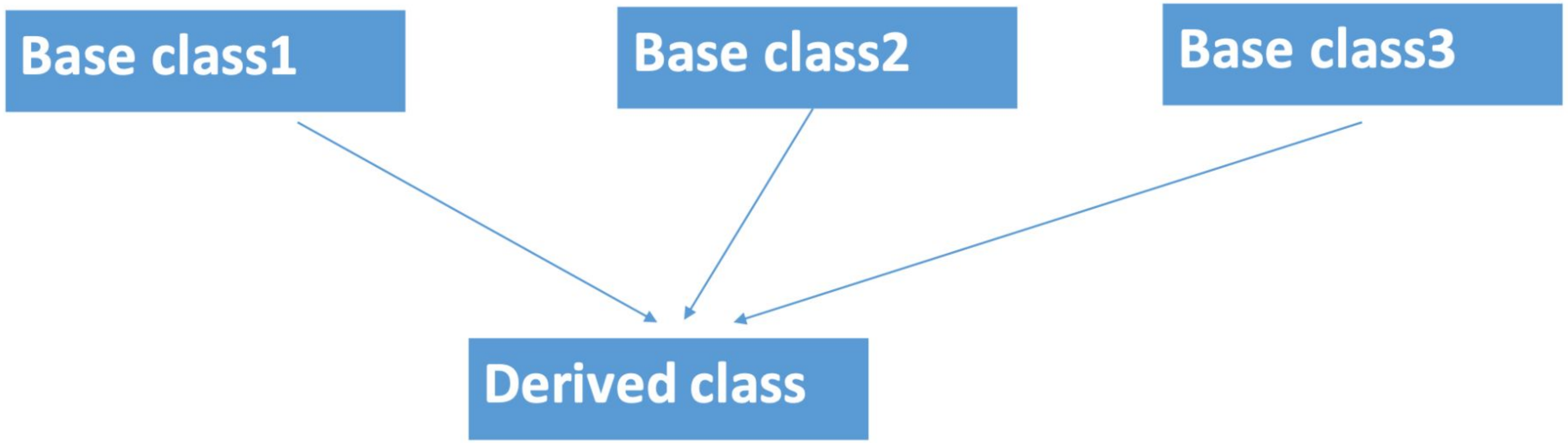
3. Hierarchical Inheritance:

- More than one child class is inherited from a base class in hierarchical inheritance



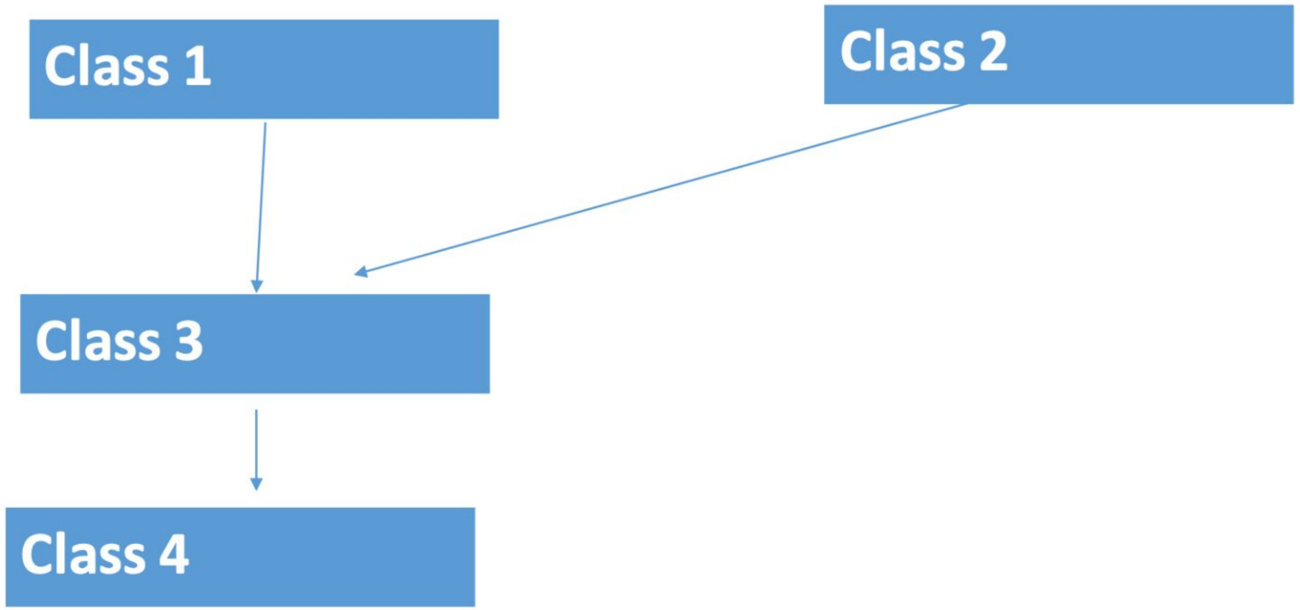
4. Multiple Inheritance:

- In this type of inheritance a child class is derived from more than one base class



5. Hybrid Inheritance:

- The inheritance which involves more than one form of inheritance is called hybrid inheritance



1. Single Inheritance:

• When a single child class is being inherited by a single parent class, it is called single inheritance

• i.e. one derived class with only one base class

Syntax to define a derived class using single inheritance:

```
class Derived_Class_Name: Inheritance_Mode Base_Class_Name {
    class body
    };
```

Example Program : using public inheritance mode

```
#include <iostream>

using namespace std;

class ABC{

        private: int a;

        protected: int b;

        public: int c;

        void setValue();

        void displayValue();

        int getA() {

                return a;

                }

        };
```

```
class XYZ: public ABC {
        private:
        int x;
        public :
        void calculate();
        void displayResult();
        };

void ABC :: setValue()     {
cout << "Enter value of a, b and c:";
cin >> a >> b >> c;
}
void ABC :: displayValue()
{
cout << "Value of a : "<<a <<endl ;
cout << "Value of b : "<<b<<endl;
cout << "Value of c : "<<c <<endl;
}
```

```
void XYZ :: calculate()

{

x = getA() + b + c;

}

void XYZ :: displayResult()

{

cout<<endl<<"Result after addition is : "<< x <<endl;

}
```

```
int main()
{
XYZ obj;
obj.setValue();
obj.displayValue();
obj.calculate();
obj.displayResult();
// obj.b =200; //protected members are not
accessible outside class.
obj.c =100; //public members are accessible
outside class
cout << endl <<"After updating the value of
public member c :"<<endl;
obj. displayValue();
}
```

Example Program : using private inheritance mode

```
#include <iostream>

using namespace std;

class ABC{

        private: int a;

        protected: int b;

        public: int c;

        void setValue();

        void displayValue();

        int getA() {

                return a;

                }

        };
```

```
class XYZ: private ABC // replace visibility mode with protected
{
private:
int x;
public :
void calculate();
void displayResult();
};

        void ABC :: setValue()
        {
        cout << "Enter value of a, b and c:";
        cin >> a >> b >> c;
        }
        void ABC :: displayValue()
        {
        cout << "Value of a : "<<a <<endl ;
        cout << "Value of b : "<<b<<endl;
        cout << "Value of c : "<<c <<endl;
        }
```

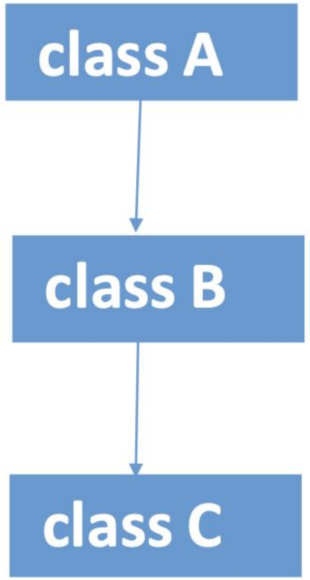
```
void XYZ :: calculate()
{
setValue(); //accessing setValue() of
Base class ABC
x = getA() + b + c;
}
void XYZ :: displayResult()
{
displayValue(); //accessing
displayValue() of Base class ABC
cout<<endl<<"Result after addition is :
"<< x <<endl;
}
```

```
int main() {
	XYZ obj;
	/* obj.setValue();
	obj.displayValue();    cannot be accessed
outside class as XYZ has inherited ABC’s
properties in private mode */
	obj.calculate();
	obj.displayResult();
	// obj.b=200; //private members are not
accessible outside class.
	//obj.c =100; //private members are accessible
outside class
```

Practice: Q1. Write a program to create a class named Person which has name and age as data members and member functions to read and display its data. Create another class Student derived from class Person to use the features of base class.

2. Multilevel Inheritance:

```
class A
        {
                body of class A
        };
class B : visibility_mode A
        {
                body of class B
        };
class C : visibility_mode B
        {
                body of class C
        };
```



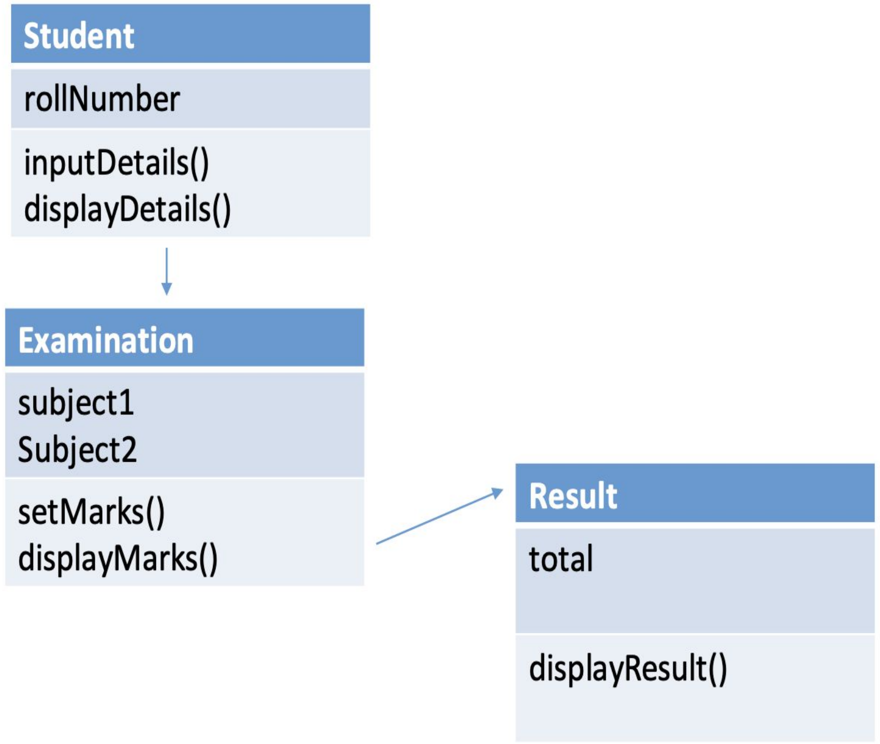
```
#include<iostream>
using namespace std;
class Base {
      protected:
      int a;
      public:
      void setData () {
            cout << "Enter the value of a = ";
            cin >> a;
} };
class DerivedOne : public Base {
      protected:
      int b;
      public:
      void readData () {
      cout << "Enter the value of b = ";
      cin >> b;
      } };
```

```
class DerivedTwo : public DerivedOne {
      private:
       int c;
      public:
            void input() {
            cout << "Enter the value of c = ";
            cin >> c;
            }
            void product() {
             cout << "Product = " << a * b * c;
      //accessing members a and b of base
      class } };
```

```
int main () {
      DerivedTwo obj;
      obj.setData(); //accessing member of Base class
      obj.readData(); //accessing member of DerivedOne
      obj.input();
      obj.product();
      return 0;
      }
```

Practice:

Q1. Define a class Student which has roll number and member functions to input and display roll number. Derive a class Examination from class Student which has marks of two subjects and member functions to initialize and display marks. Again derive a class Result from class Examination and calculate total and display the result.

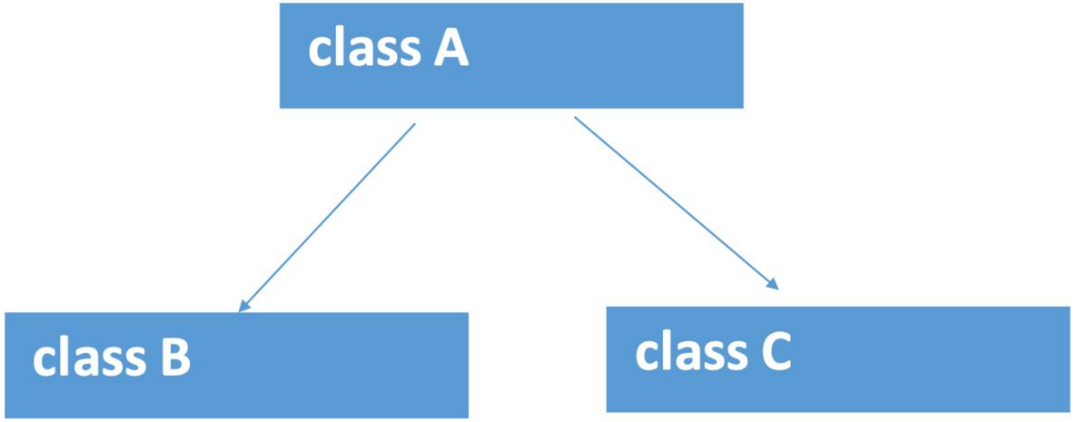


```
#include <iostream>

using namespace std;
class Student{
    protected:
    int roll;
    public:
    void input(){
        cout<<"Enter details:";
        cin>>roll;
    }
    void display(){
        cout<<"Roll no: "<<roll<<endl;
    }
};
class Examination: public Student{
    protected:
    int subject1,subject2;
    public:
    void set_marks(){
        cout<<"Input marks of subjects:";
        cin>>subject1>>subject2;
    }
    void display_marks(){
        cout<<"Marks of subject1: "<<subject1<<endl;
        cout<<"Marks of subject2: "<<subject2<<endl;

    }
};
class Result: public Examination{
    private:
    int total;
    public:
    void displayResult(){
        input();
        cout<<"Mark details of roll no. "<<endl;
        display();
        cout<<"Marks of individual subjects are: "<<endl;
        set_marks();
        display_marks();
        total = subject1+subject2;
        cout<<"Marks obtained: "<<total<<endl;
    }
};
```

```
int main()
{
    Result r;
    r.displayResult();

    return 0;
}
```

Hierarchical Inheritance

```
class A
        {
                ...
        };
class B : visibility_mode A
        {
                ...
        };
class C : visibility_mode A
        {
                ...
        }
```



```
#include <iostream>
using namespace std;
class Base //single base class
{
      protected:
     int a,b;
     public:
     void setData () {
     cout << "Enter the value of a : ";
     cin >> a;
     cout << "Enter the value of b : ";
     cin >>b;
     } };
```

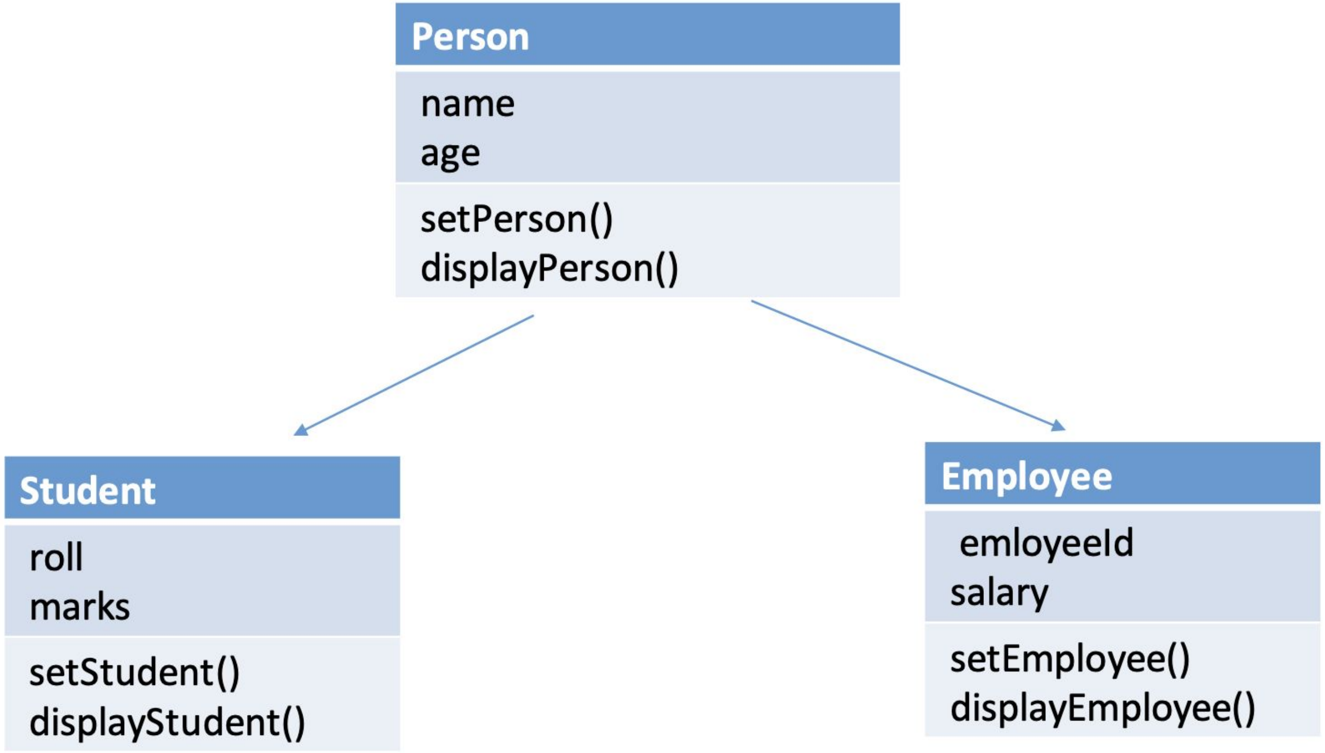
```
class DerivedOne : public Base
//DerivedOne is derived from class Base
{
protected:
int c;
public:
void add()
{
c=a+b; //access the member of base class
cout << a << "+" << b <<"="<<c<<endl;
}
};
```

```
class DerivedTwo : public Base
//DerivedTwois also derived from class Base
{
private:
int c;
public:
void product()
{
c=a*b;
cout << a < < "*" << b <<"="<<c<<endl;
}
};
```

```
int main () {
DerivedOne obj1; //Object of DerivedOne
class
DerivedTwo obj2; //Object of DerivedTwo
class
obj1.setData(); // call member function of
Base
obj1.add();
obj2.setData(); // call member function of
Base
obj2.product();
return 0;
}
```

Practice

Q1. Write a program to use hierarchical inheritance.

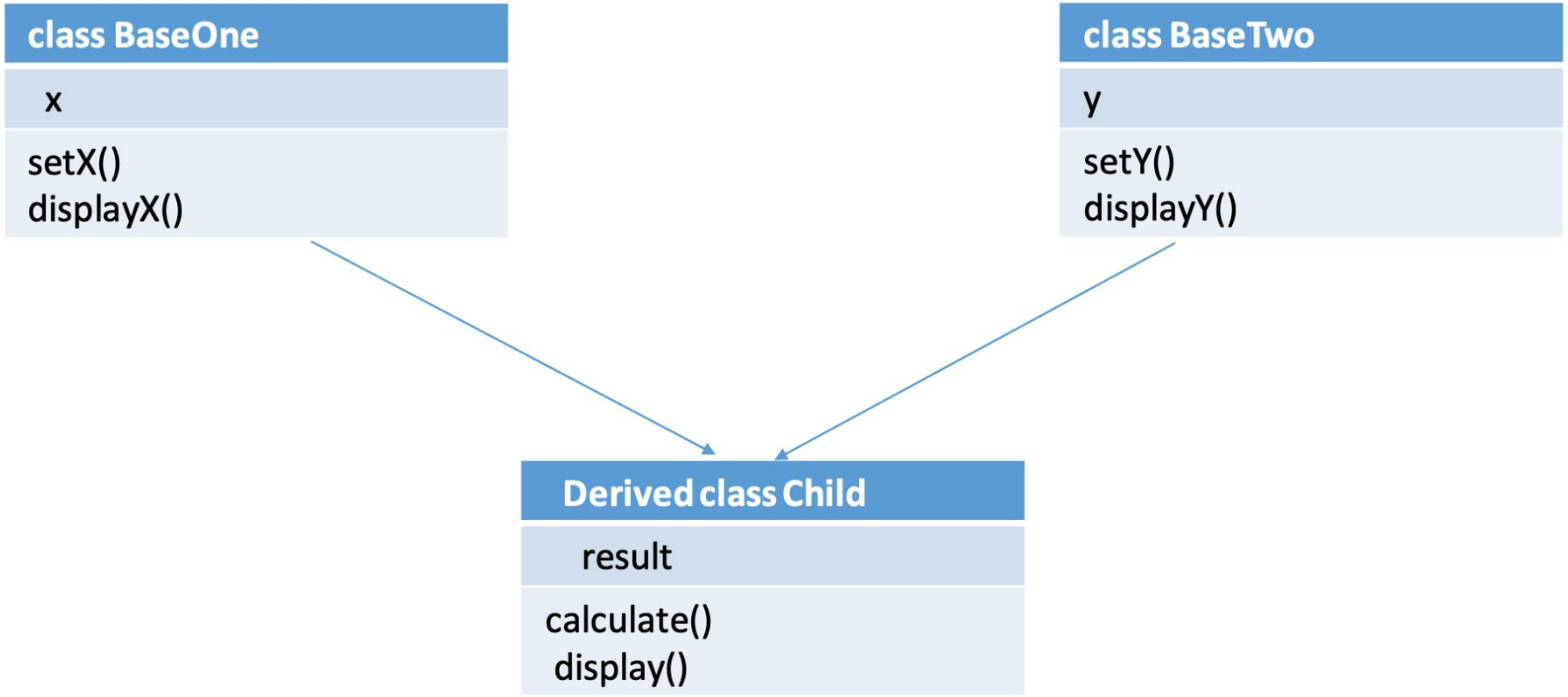


4. Multiple Inheritance:

Syntax to define a derived class using multiple inheritance:

```
class Base_Class_One
        {
        ...
        };
class Base_Class_Two
        {
                ...
        };
class Derived_Class : visibility_mode Base_Class_One , visibility_mode Base_Class_Two
        {
          ...
        };
```

Example :


class BaseOne
x
setX()
displayX()
class BaseTwo
y
setY()
displayY()
Derived class Child
result
calculate()
display()

```
#include <iostream>
using namespace std;
class BaseOne {
      protected :
      int x;
      public :
       void setX();
      void displayX();
};
class BaseTwo {
      protected :
            int y;
            public :
            void setY();
            void displayY();
};
```

```
void BaseOne:: setX()
{
cout << "Enter value of x :";
cin >> x ;
}
void BaseOne :: displayX()
{
cout << endl <<"The value of x is : " <<x
<<endl;
}
void BaseTwo:: setY()
{
cout << "Enter value of y :";
cin >> y ;
}
void BaseTwo :: displayY()
{
cout << endl <<"The value of y is : " <<y
<<endl;
}
```

```
class Child : public BaseTwo, public BaseOne {
	private:
		int result;
	public:
	void calculate();
	void display();
};
void Child :: calculate()
{
result = x + y; //accessing x and y inherited from
its base classes
}
void Child :: display()
{
cout << endl <<"The result after addition is
:"<<result <<endl;
}
```

```
int main() {

	Child obj;
	obj.setX();
	obj.setY();
	obj.displayX();
	obj.displayY();
	obj.calculate();
	obj.display();

}
```

Ambiguity in Multiple Inheritance

- Ambiguity arises in multiple inheritance when more than one base class have same function name which is not overridden in derived class.
- If we try to call the function using the object of the derived class, compiler shows error because compiler doesn't know which function to call

```
#include <iostream>
using namespace std;
class BaseOne {
      protected :
            int x;
      public :
      void set();
      void display();
};
class BaseTwo {
      protected :
      int y;
      public :
      void set();
      void display(); };
```

```
void BaseOne:: set() {
        cout << "Enter value of x :";
        cin >> x ;
        }
void BaseOne :: display() {
cout << endl <<"The value of x is : " <> y ;
 }
void BaseTwo :: display()
{
cout << endl <<"The value of y is : " <<y <<endl;
}
void Child :: calculate()
{
result = x + y;
}
void Child :: displayResult()
{
cout << endl <<"The result after addition is :"<<result
<<endl;
}
```

```
class Child : public BaseTwo, public
BaseOne {
      private:
      int result;
       Public:
       void calculate();
      void displayResult();
};
```

```
int main() {
      Child obj;
      obj.set(); /*compiler throws an error as
      there is ambiguity about which base
      class's set() to call */ obj.set();
      obj.display(); / *compiler throws an
      error as there is ambiguity about which
      base class's display() to call */
      obj.display();
      obj.calculate();
      obj.displayResult();
      }
```

Function Overriding

We use base class name and scope resolution operator to call base class version using object of derived class.

```
#include<iostream>
using namespace std;
class Base
{
public :
void display()
{
cout<<"This is base "<<endl;
}
};
class Derived: public Base
{
public :
void display()
{
cout<<"This is derived "<<endl;
}
};
```

```
int main()
{
Base bobj;
cout<<"Calling from Base class's
object"<<endl;
bobj.display();
Derived dobj;
cout<<endl<<endl<<"Calling from Derived
class's object"<<endl;
dobj.display();
dobj.display();

}
```

```
#include<iostream.h>
#include<conio.h>
class Base
{
        public :
                void display()
                {
                        cout<<"This  is base "<<endl;
                }
};

class Derived: public Base
{
        public :
                void display()
                 {
                        cout<<"This  is derived "<<endl;
                }
};
```

```
void main()
{
        Base bobj;
        cout<<"Calling from Base class's object"<<endl;
        bobj.display();
        Derived  dobj;
        cout<<endl<<endl<<"Calling  from Derived
class's  object"<<endl;
        dobj.Base::display();
        dobj.display();
        getch();
}
```

Function Overriding

- Two or more functions having same name and same signature (number, type and sequence of parameters) but one defined in base class and other defined in derived class
- In function overriding, the compiler doesn’t know about the correct form of function to be called in compile time. The correct form of function is selected to call on the basis of content of calling object at runtime or while the program is running.
- The code associated with the function call is not known until program execution, hence called late binding/ dynamic binding/ dynamic linkage.

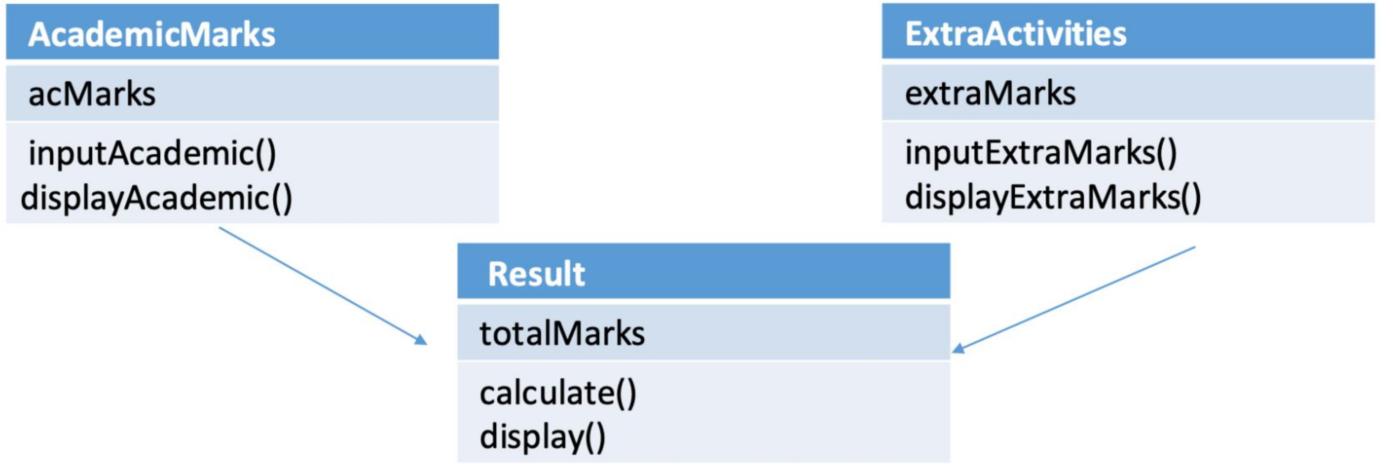
Solution of ambiguity in multiple inheritance:

The ambiguity can be resolved by using the class_name and scope resolution operator (::) to specify the class name of member function.

```
int main() {
        Child obj;
        obj.BaseOne::set();             //calls BaseOne'sset()
        obj.BaseTwo::set();             //calls BaseTwo'sset()
        obj.BaseOne::display();         //calls BaseOne'sdisplay()
        obj.BaseTwo::display();               //calls BaseOne'sdisplay()
        obj.calculate();
        obj.displayResult();
        }
```

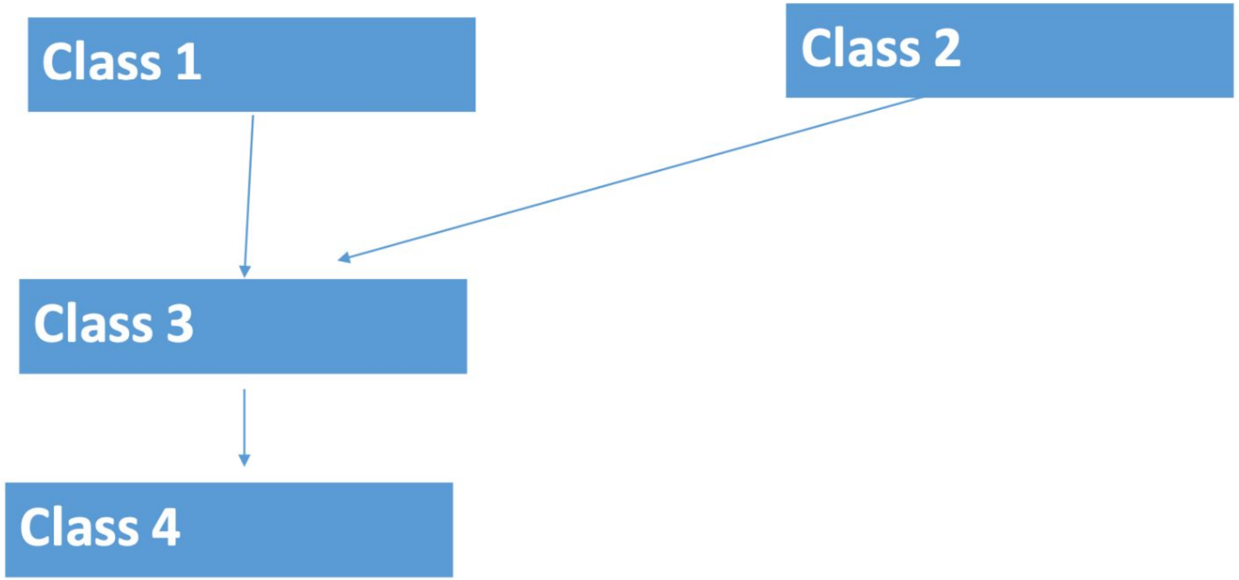
Practice:

Q1. Write a program to define class AcademicMarks and ExtraActivities which have data members for academic marks and extra activities marks respectively. Define member functions to initialize the marks. Derive a class Result from AcademicMarks and ExtraActivities and calculate the total marks obtained.



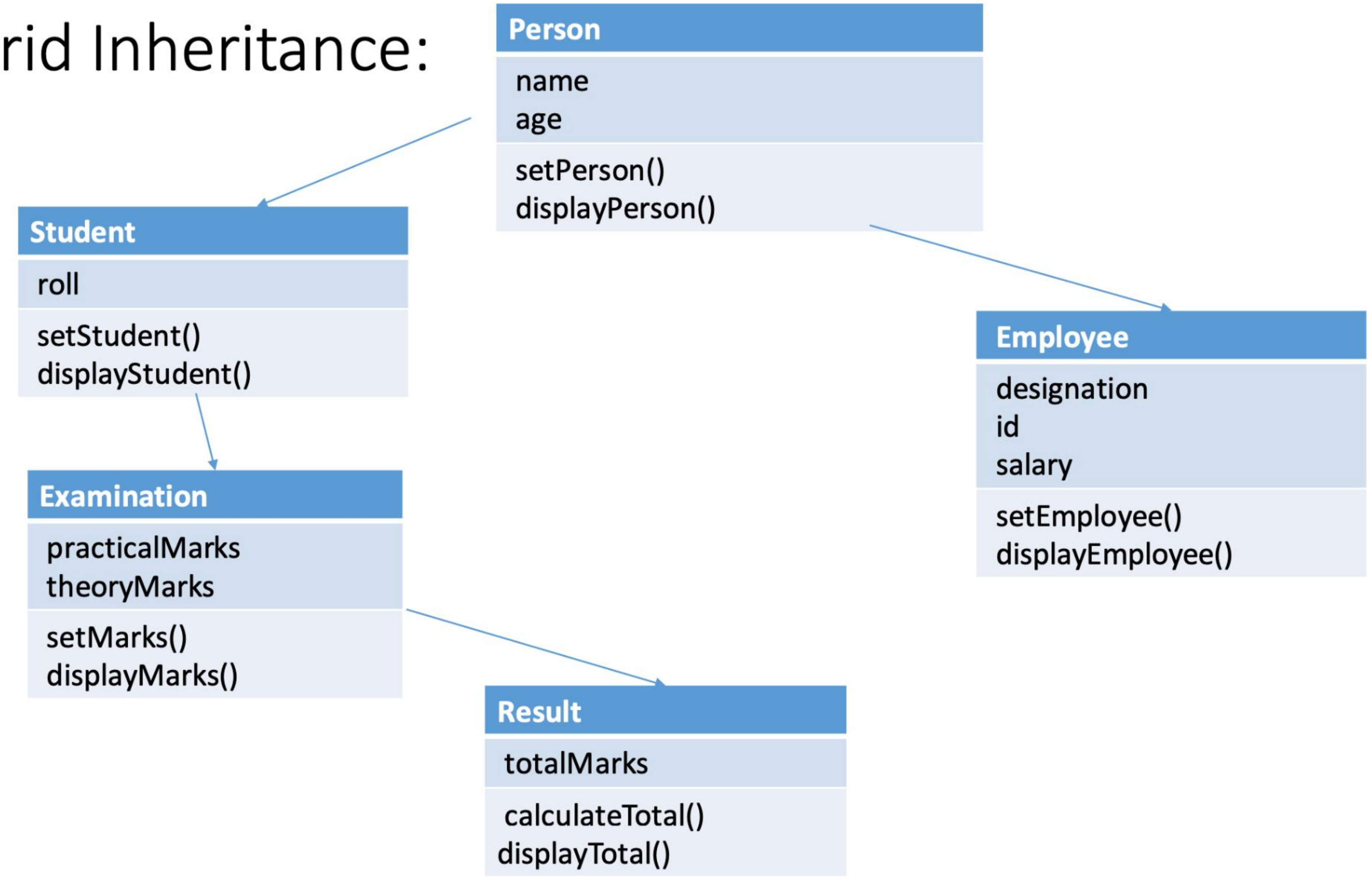
5. Hybrid Inheritance:

- The inheritance which involves more than one form of inheritance is called hybrid inheritance



Hybrid Inheritance:

Q1.



```
#include <iostream>
using namespace std;
class Person{
    protected:
    string name;
    int age;
    public:
    void setPerson(){
        cout<<"Enter name and age:";
        cin>>name>>age;
    }
    void displayPerson(){
        cout<<"Name: "<<name<<endl;
        cout<<"Age: "<<age<<endl;
    }
};
```

```
class Student:public Person{
    protected:
    int roll;
    public:
    void setStudent(){
        cout<<"Enter roll:";
        cin>>roll;
    }
    void displayStudent(){
        cout<<"Roll: "<<name<<endl;
    //    cout<<"Age: "<<age<<endl;
    }

};
```

```
class Employee: public Person{
    int id;
    float salary;
    string designation;
    public:
    void setEmployee(){
        cout<<"Enter id,salary, designation:";
        cin>>id>>salary>>designation;
    }
    void displayEmployee(){
        cout<<"Id: "<<id<<endl;
        cout<<"salary: "<<salary<<endl;
        cout<<"designation: 
"<<designation<<endl;
    }

};
```

```
class Examination: public Student{
    protected:
    int practical,theory;
    public:
    void setMarks(){
        cout<<"Enter practical and theory 
marks:";
        cin>>practical>>theory;
    }
    void displayMarks(){
        cout<<"practical marks: 
"<<practical<<endl;
        cout<<"theory marks: 
"<<theory<<endl;
    }
};
```

```
class result: public Examination{
    int totalmarks;
    public:
    void calculateTotal(){
        setPerson();
        setStudent();
        setMarks();
        totalmarks=practical+theory;
    }
    void displayTotal(){
        displayPerson();
        displayStudent();
        displayMarks();
        cout<<"Total Marks: "<<totalmarks<<endl;
    }
};
```

```
int main()
{
    result r;
    r.calculateTotal();
    r.displayTotal();

    return 0;
}
```

Practice

Q1. Create a class called Student with two data member to represent name and age of the student. Use member function to read and print these data. From this class, derive a class called Boarder with data member to represent room number. Derive another class called DayScholar from the class Student with data member to represent address and bus number of the student. In both derived classes, use member function to read and print the respective data.

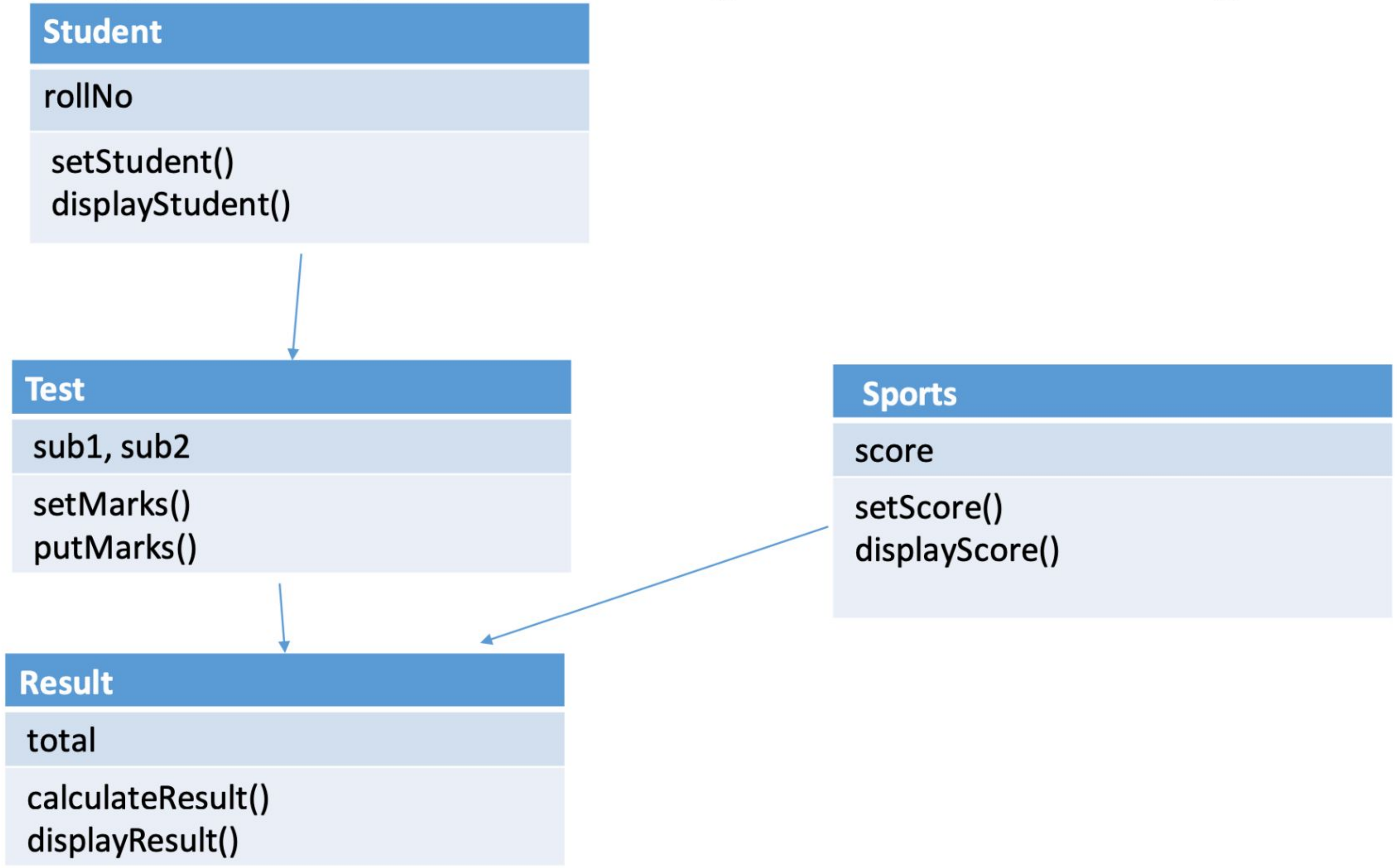
Practice

Q2. Create a class called Rectangle with data member to represent length, breadth and area. Use appropriate member functions to read length and breadth from user to calculate area of the rectangle. Next create a derived class called Box from the class Rectangle. Use appropriate member function to read height and to calculate the volume of the box.

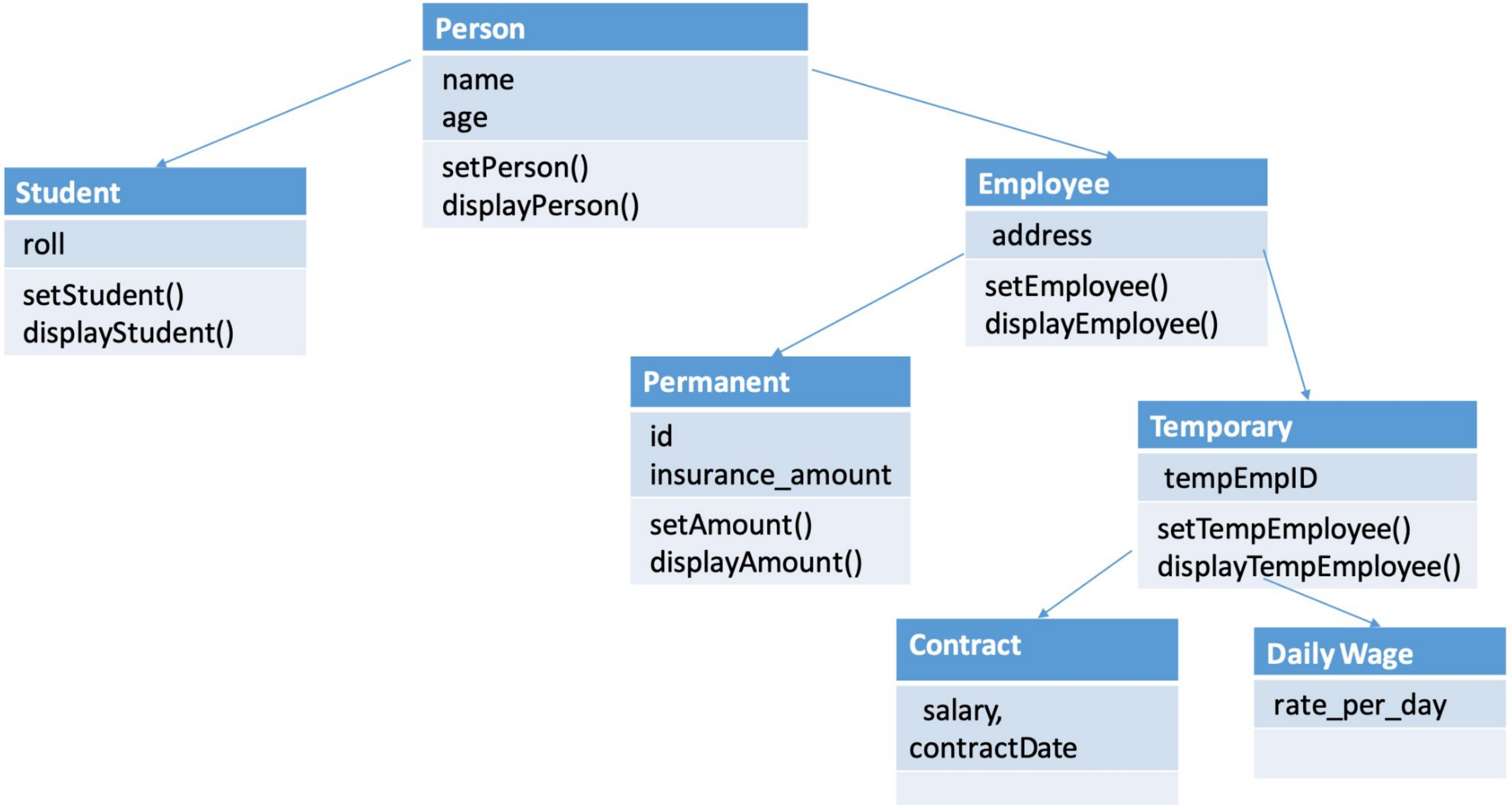
Practice

Q3. Write a base class that ask the user to enter a complex number and derived adds the complex number of its own with the base. Finally make third class that is friend of derived and calculate the difference of base complex number and its own complex number.

Q4. Write a program to represent following classes with following data members.



Q6. Write a program to represent following classes.



Virtual Function

In C++, function overriding is achieved using virtual function.

A virtual function is a member function which is declared within a base class and overridden by derived class

When we use same function name with same signature in base class and derived class, the function in base is declared as virtual function

When the class containing virtual function is inherited, the derived class redefines virtual function to perform task respective to derived class

Virtual Function

Base class pointer can point object of same class and also point object of its derived classes

The correct version of function is called on the basis of type of object pointed by base class pointer at particular time

By making the base class pointer point to different class’s object, we can execute different version of virtual function

If the derived class doesn’t redefine virtual function, the derived class inherits its immediate base class virtual function definition

Base class pointer can point to object of any of its descendant class

```
#include<iostream>
using namespace std;
class Base
{
public :
void display()
{
cout<<"This is base "<<endl;
}
};
class Derived: public Base
{
public :
void display()
{
cout<<"This is derived "<<endl;
}
};
```

```
int main(){
Base *bptr, bobj;
bptr=&bobj;
cout<<"Calling from Base class's pointer that
points to Base's object"<<endl;
bptr->display();
Derived dobj;
bptr=&dobj;
cout<<endl<<endl<<"Calling from Base
class's pointer that points to Derived's
object"<<endl;
bptr->display();
}
```

```
#include<iostream>
using namespace std;
class Base
{
public :
virtual void display()
{
cout<<"This is base "<<endl;
}
};
class Derived: public Base
{
public :
void display()
{
cout<<"This is derived "<<endl;
}
};
```

```
int main(){
Base *bptr, bobj;
bptr=&bobj;
cout<<"Calling from Base class's pointer that
points to Base's object"<<endl;
bptr->display();
Derived dobj;
bptr=&dobj;
cout<<endl<<endl<<"Calling from Base class's
pointer that points to Derived's object"<<endl;
bptr->display();
}
```

Virtual Constructor: • Constructor cannot be virtual because when constructor of a class is executed, virtual pointer is not defined yet.

Virtual Destructor: • A virtual destructor is used to free up the memory space allocated by the derived class object while deleting instance of the derived class using base class pointer.

Pure Virtual function:

- Virtual function which has no body is known as pure virtual function
- It is a virtual function for which we do not need to write any function definition, we only need to declare it

Syntax to declare pure virtual function: virtual returnType functionName()=0;

- It is used when function doesn’t have any use in the base class but must be implemented by all its derived class

Pure Virtual function:

- Pure virtual is declared in base class and it has no function definition relative to the base class
- Each derived class define the virtual function as per its requirement

Abstract Class:

- It is a class that has at least one pure virtual function
- The classes inheriting the abstract class must provide the definition for the pure virtual function otherwise the subclass would become an abstract class itself
- Abstract class cannot be instantiated, but pointers of abstract class can be created
- Abstract class can have normal functions and variables along with a pure virtual function

Programs:

1. Define a class Person with attributes and behaviors that are common to Student and Teacher, Person class doesn’t have any implementation of the functions, so make it abstract class. Derive class Student and Teacher from class Person and provide the implementation of the inherited functions in each derived class.

Subclass, Subtype and Substitutability

The relationship of the data type associated with a parent class to the data type associated with a child class gives rise to the following arguments:

i. Instances of the subclass must possess all data areas associated with parent class

ii. Instances of the subclass must implement thru inheritance at least all functionality defined for the parent class (They can also define new functionality)

iii. An instance of a child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent class if substituted in a similar situation

Principal of Substitutability :

- “If we have two classes A and B such that class B is a subclass of A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect”

```
#include<iostream>
using namespace std;
 class A
{
public:
void printMsg()
{
   cout <<" I am A!!";

} };
```

```
class B : public A {
public:
void printMsg()
{
cout <<" I am B!!";
}
};
void testingA(A a) {
a.printMsg ();
}
void testingFunc() {
   B b;
cout<<" Passing B"<<endl;
testingA(b);    // b substituted for object of A.
}
int main() {
testingFunc();
}
```

Subclass:

- A class which inherits the characteristics and behavior of another class is called a subclass.

Subtype:

- Subtype is subclass relationship in which the principle of substitutability is maintained

IS-A Relationship

- In OOP “is-a” relationship is a relation between two classes where one class is a specialized form of second class
- X is a specialized instance of concept Y if the assertion in plain English “X is a Y” sounds correct or logical
- Examples of inheritance satisfying the “is-a” relationship • student is a person, apple is a fruit, car is a vehicle
- The concept of “is-a” relationship is represented by Inheritance

HAS-A Relationship

- If the one concept is a component of another concept then there exist “has-a “ relationship
- the two concepts are not the same thing
- e.g. a car has a engine , fire station has fire fighter
- Also called a “part of” relationship e.g. an engine is a part of car
- Testing the relationship:
  - Form a simple sentence “ X has a Y”,
  - if the result sounds reasonable then the relationship hold

Example of “Car has a engine”:

```
class Engine {


};

class Car

{

Engine e;                    // a car is composed of an engine

};
```

```
#include<iostream>
using namespace std;
class Engine
{
public:
void startEngine()
{
   cout<<" Engine started"<<endl;
}
void stopEngine() {
cout<<" Engine Stopped"<<endl;
} };
```

```
class Car {

Engine e;
public:
void startCar() {
e.startEngine();
cout<<endl<<" The car is moving"<<endl;
}
void stopCar() {
e.stopEngine();
cout<<endl<<" The car has
stopped"<<endl;
}
};
```

```
int main(){
    Car car;
car.startCar();
cout<<endl<<endl<<endl;
car.stopCar();

}
```

Composition

Composition is an alternative to the class inheritance that serves different purposes. In composition, the models have a “has a relationship”.

It lets you create complex types by combining behaviors and characteristics of other types.

In a class created with composition, there is a composite side, which is a collection of component instances, and a component side, which are the instances the composite class will contain.

Composition appears to occur when an object contains another object and the contained object cannot exist without the existence of the other object.

Inheritance:

1. One object acquires characteristics of one or more other objects
2. Class inheritance is defined at run-time
3. Exposes both public and protected base classes
4. No access control
5. Often breaks encapsulation

Composition:

1. Using an object within another object.
2. Defined dynamically at run-time.
3. Internal details are not exposed to each other - they interact through public interfaces.
4. Access can be restricted
5. Won’t break encapsulation