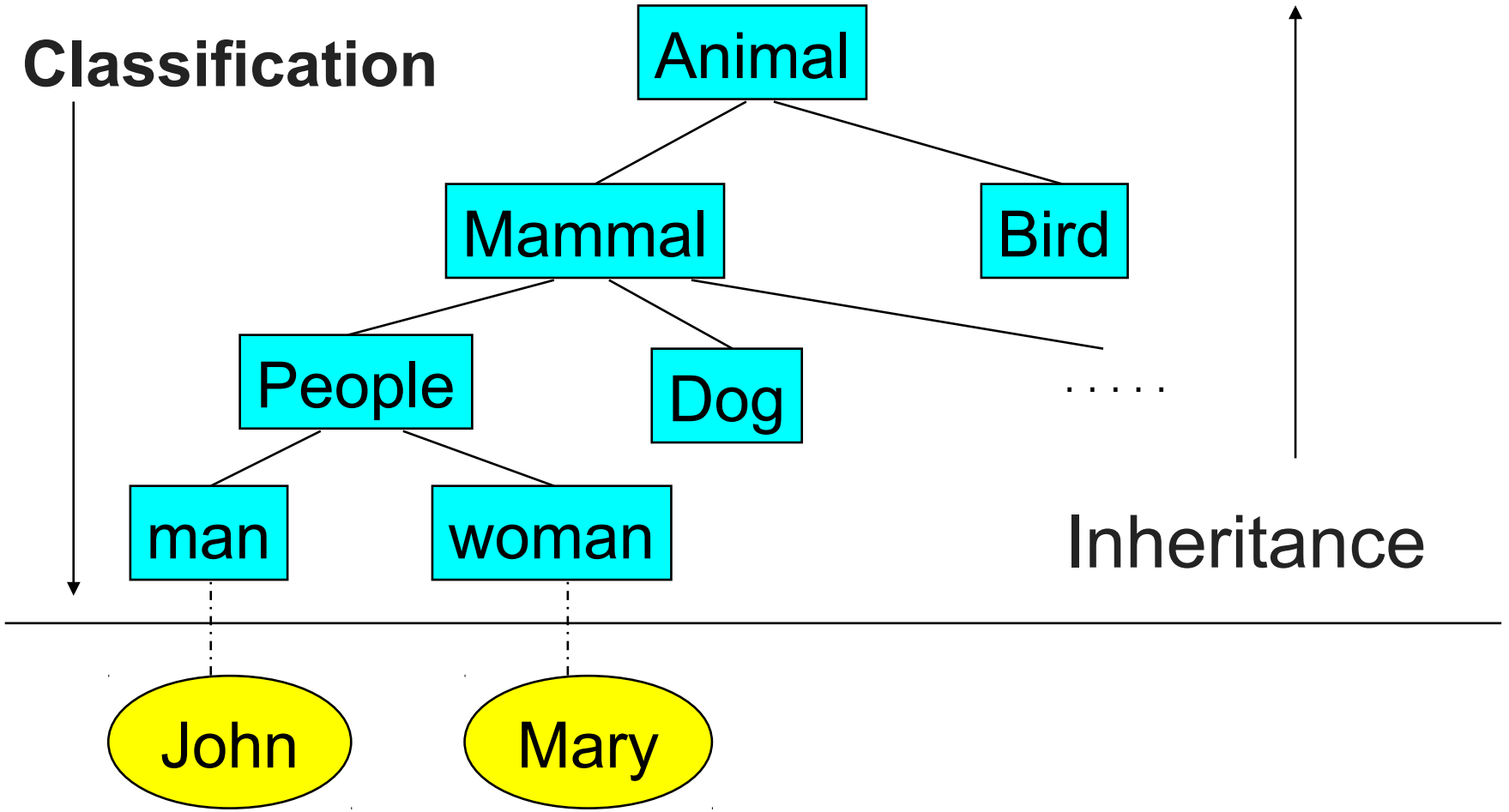
Inheritance

Classification/Inheritance



Classification and inheritance

- Classification arises from the universal need to describe uniformities of collection of instances.
- Classification is a basic idea for understanding the concept inheritance
- Inheritance is the technique to realize sharing of information without explicit redefinition.
- Inheritance means that new classes can be derived from existing classes.

Classification and inheritance(2)

- The subclass inherits the attributes and the operations of the superclass (base class).
- In addition, the subclass (derived class) can define additional operations and attributes.
- It is a specialization mechanism.
  - Instances of a subclass are specializations of the instances of the superclass.
- It is also a generalization mechanism.
  - Instances of the superclass generalize (restrict) the instances of the subclasses.

Why we use inhertiance?

- It promotes software reuse by allowing the defining of a hierarchy of classes
  - with successive refinement of behavior
  - additional behavior
  - restrictions on behavior
  - additional data members
- Allows using existing class libraries and tailoring them to the current situation

Using Inheritance

- Imaginary goal: Design a class Dog.
- Don't start from scratch if a very similar class, e.g. Mammal, already exists.
- Inherit from an existing class instead. Terminology: Dog inherits from Mammal, Mammal is the superclass and Dog is the subclass.

Why not Copy-and-paste?

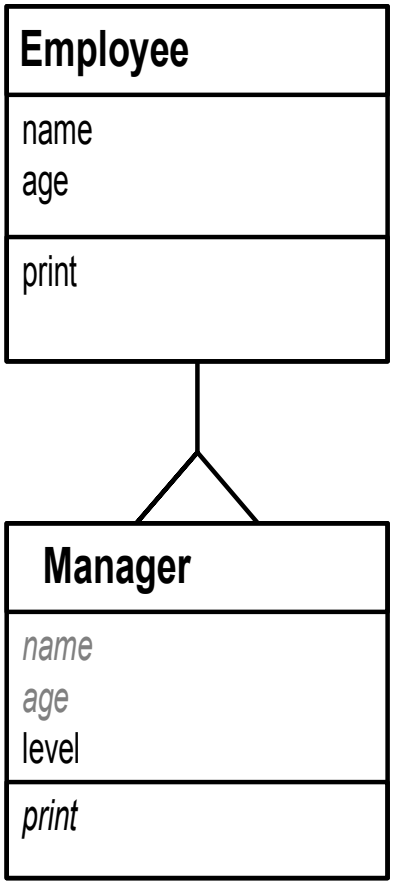
- Copy the Mammal class Paste it
- Change name to Dog
- Modify some existing methods, add some new methods
- Fine, but..
  - Get two copies of similar code, then change it;
  - DANGER for future debugging Code: the size is blown up!!

Definition : inheritance

- **Inheritance** is a mechanism for expressing similarity.
- **Inheritance** is the natural property of classification or Hierarchy.
- **Inheritance** is the mechanism which allows one class to inherit properties and behaviours of another class.
- **Inheritance** is the technique to define base class – derived class relationship

Superclass/subclass

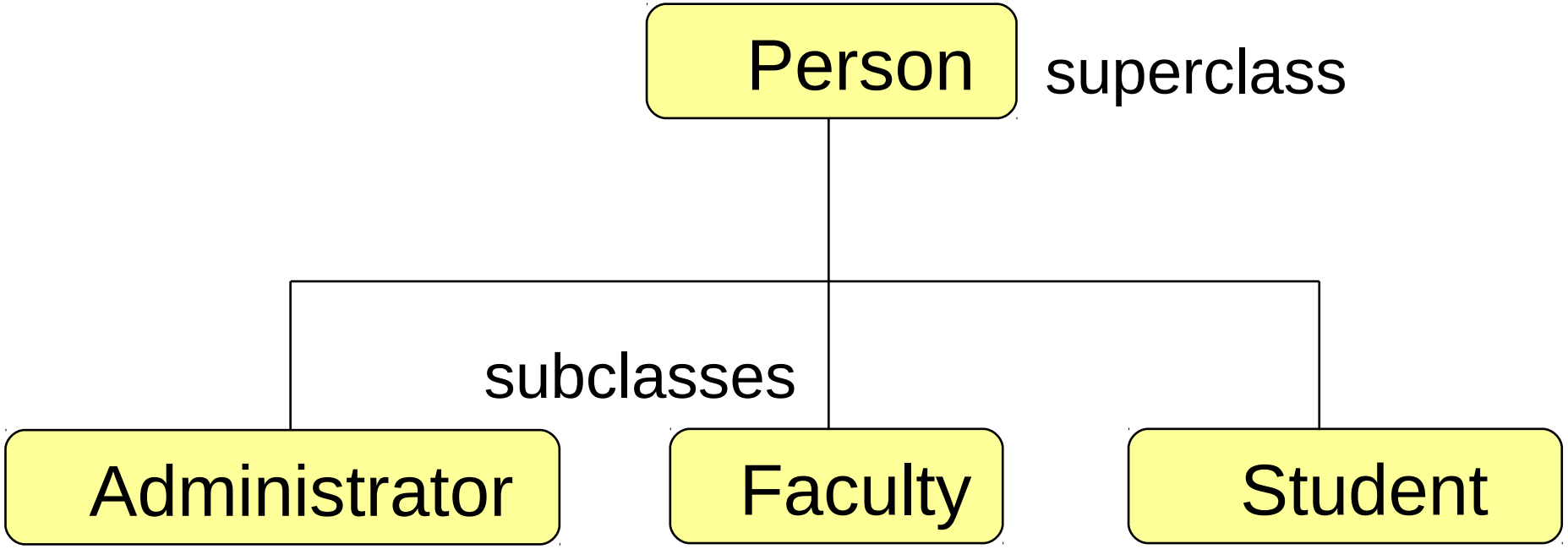
- If class A inherits from class B then
  - B is called superclass (base class) of A.
  - A is called subclass (derived class) of B.
- Objects of a subclass can be used where objects of the corresponding superclass are expected, because objects of the subclass share the same behavior as objects of the superclass.



Superclass/
Base class

Derived class/
Sub-class

Example


Person
superclass
subclasses
Administrator
Faculty
Student

Inheritance format

```
class derived-class-name : access base-class-name {
    // body of class
};
```

- An inheritace access type of a derived class from a base class can be either of the following way
  - Public
  - Protected
  - Private
- access specifier is private by default

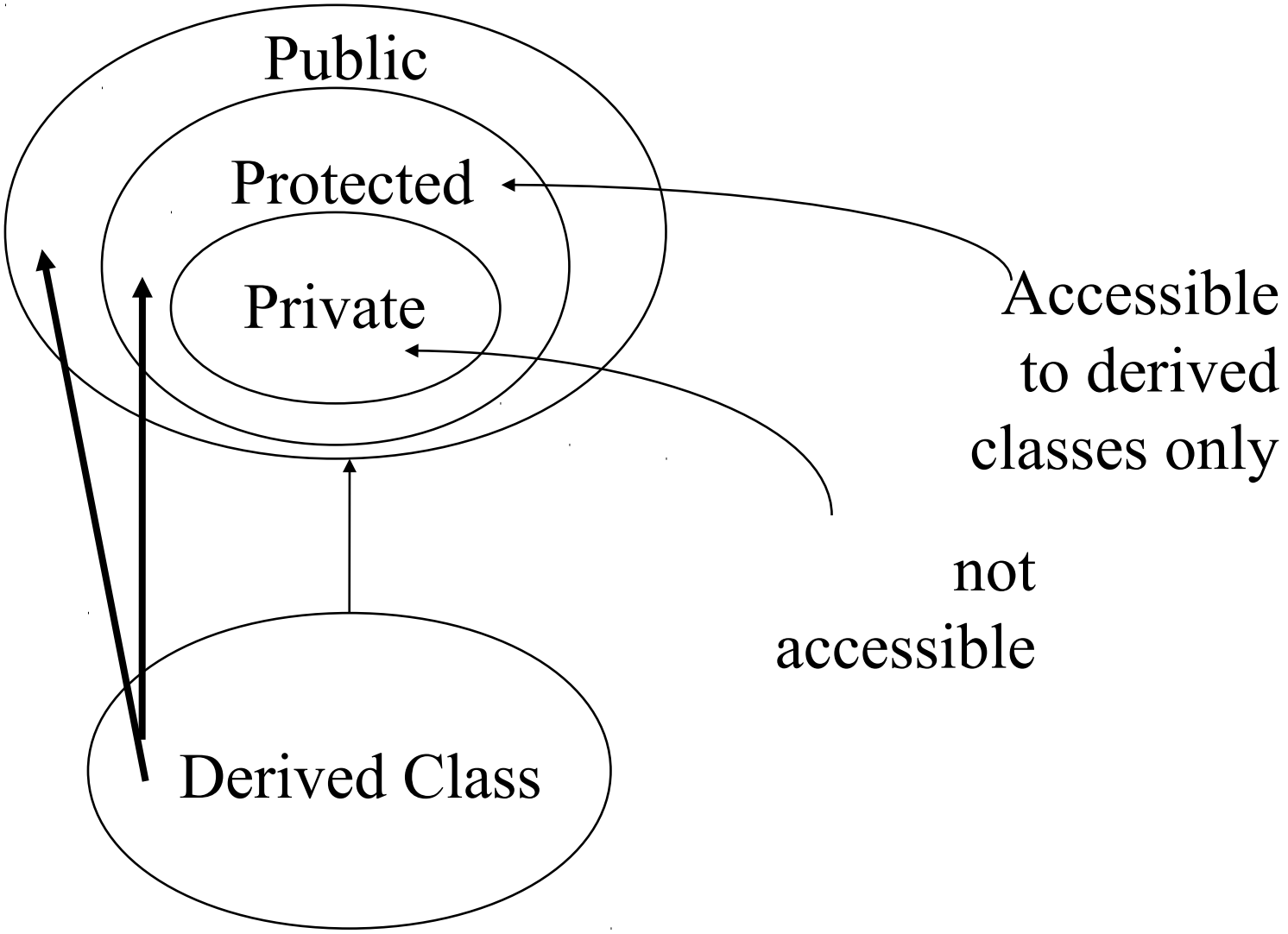
Public inheritance

- When we use public inheritance
  - Private members (either data or functions) of a base class are not accessible to derived classes
  - However, they are accessible through public member functions of the base class.
  - Public members of a base class are accessible to the derived class.

Protected inheritance

- Protected members of the base class can only be accessed by members and friends of the base class, and any derived class.
- Protected data violates data encapsulation,
  - changes to the base class protected members may require changes to all derived classes.

Protected inheritance



Visibility in inheritance types

Inheritance Type	Base class member access Specifier		
	Public	**Protected**	**Private**
Public	Public in derived class	Protected in derived class	Hidden in derived class
Protected	Protected in derived class	Protected in derived class	Hidden in derived class
Private	Private in derived class	Private in derived class	Hidden in derived class

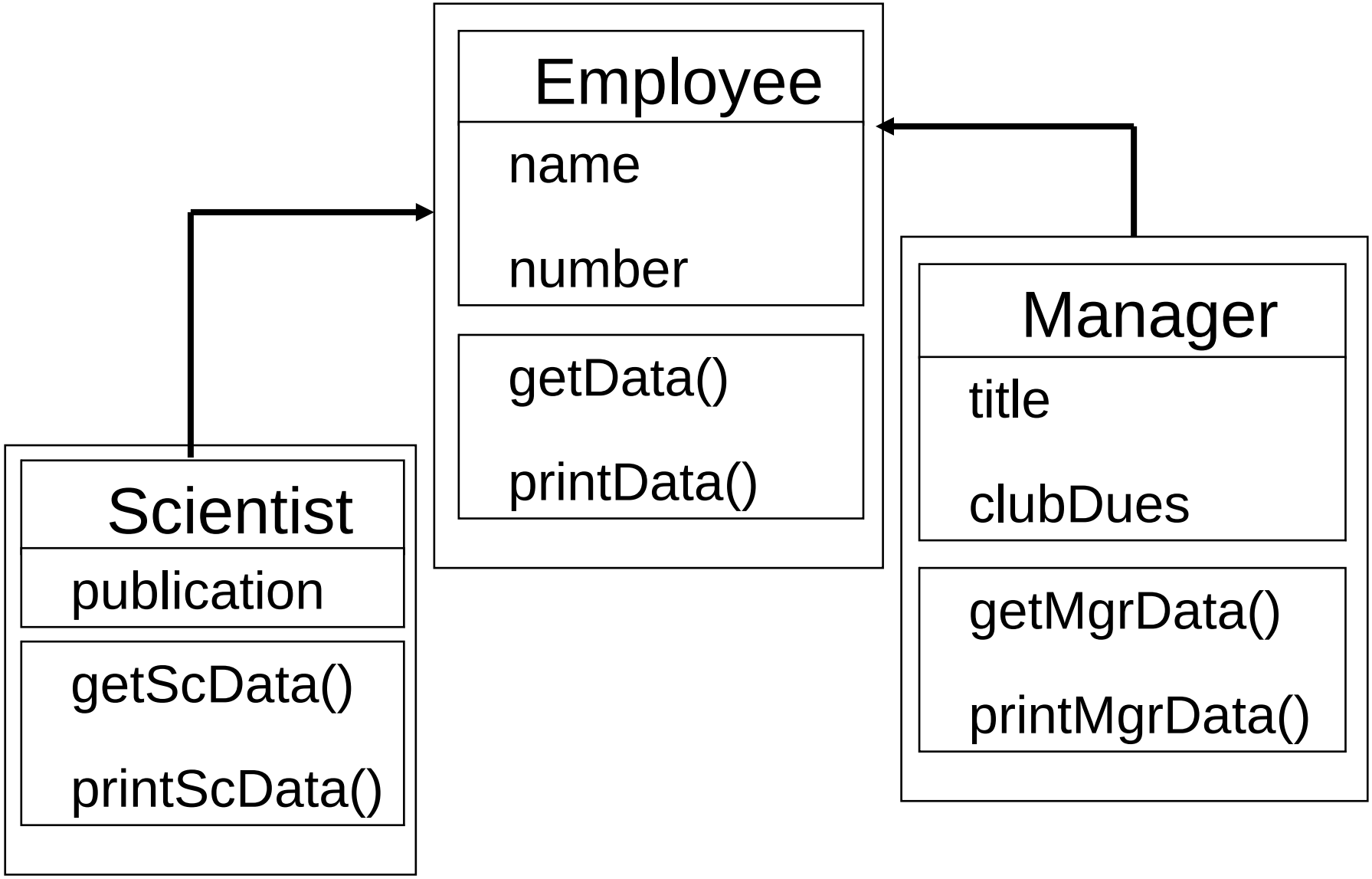
What is inherited from the base class?

- In principle, a derived class inherits every member of a base class except:
  - its constructor and its destructor
  - its operator=() members
  - its friends
- Although the constructors and destructors of the base class are not inherited themselves, its default constructor and destructor are always invoked automatically when a new object of a derived class is created or destroyed.

Category of inheritance

- Single(Simple) inheritance:
  - A class inherits from only one superclass.
  - It can be again two types
    1) Single Level
    2) Multi Level
  - The second type of inheritance hierarchy forms a tree.
- Multiple inheritance:
  - A class can have more than one super classes.

Example (Single Level)



Example(single level)

```
class Employee{
        char name[20];
        int number;
    public:
         void getData(){
            cin>>name;cin>>number;
          }
          void printData(){
              cout<<name<<number<<endl;
          }
    };
```

Example(single level)

```
class Manager:public Employee{
     char title[10];
     double dues;
    public:
      void getManagerData(){
                cin>>title; cin>>dues;
       }
       void printManagerData(){
                cout<<title<<dues<<endl;
      }
};
```

Example(single level)

```
class Scientist:public Employee{
        int pub;
  public:
         void getScData(){
                  cin>>pub;
       }
     void printScData(){
                  cout<<pub<<endl;
       }
   };
```

Example(single level)

```
main(){
        Employee e1;
        Manager m1;
        Scientist s1;
            e1.getData();
            e1.printData();
            m1.getData();
            m1.printData();
            m1.getManagerData();
            m1.printManagerData();
            s1.getData();
            s1.printData();
            s1.getScData();
            s1.printScData();
 }
```

Extending accessibility to derived class

```
class Employee{
  protected:
    char name[20];
    int number;
  public:
    void get(){
          cin>>name;
          cin>>number;
    }
    void print(){
          cout<<name<<number<<endl;
    }
};
```

Example (contd.)

```
class Manager:public Employee{
    char title[10];  double dues;
  public:
    void getData(){
            cin>>name; cin>>number;
            cin>>title; cin>>dues;
    }
    void printData(){
            cout<<name<<number<<endl;
            cout<<title<<dues<<endl;
    }
};
```

```
class Scientist:public Employee{
    int pub;
  public:
    void getData(){
            cin>>name>>number>>pub;
    }
    void printData(){

            cout<<name<<number;
            cout<<pub<<endl;

    }
};
```

Example (contd.)

```
main(){
        Employee e1;
        Manager m1;
        Scientist s1;

    e1.get ();                  e1.print();
    m1.getData();               m1.printData();
    s1.getData();               s1.printData();
    }
```

Overriding

- If attributes or methods defined in a subclass have the same name as attributes or methods defined in the super class then a technique called '**overriding**' is used to solve the conflict**.**
- The function in the derived class with the same function (or variable) name will override the functions (or variables) in the base classes.
- But it can still invoke or get the override functions(variables) with scope resolution operator (::).

Overriding(2)

- When a base class member function is overridden in a derived class, it is common to invoke the base class version and some extra work is done.
- A derived class can override a base-class member function by supplying a new version of that function with same arguments.
- Not using the scope resolution operator to reference the overridden base class member function will cause in infinite recursion.

Example : Overriding

```
class Employee{
    public:
        Employee(char*, char*);
        void print(void);
        ~Employee();
    private:
        char* fstName;
        char* lstName;
};
Employee::Employee(char* fst,char* lst){
      fstName=new char[strlen(fst)+1];
      strcpy(fstName, fst);
      lstName=new char[strlen(lst)+1];
      strcpy(lstName, lst);
}
void Employee::print(void){
      cout<<fstNam<<‘ ‘<<lstName<<endl;
}
Employee::~Employee(){
     delete []fstName; delete[]lstName;
}
```

Example : Overriding(2)

```
class HrlyWorker:public Employee {
    public:
        Hrlyworker(char*, char*, double, double);
        double getPay();
        void print(void);
    private:
        double wage;
        double hours;
};
HrlyWorker:: HrlyWorker(char* fst,char* lst, double hrs,
double wg): Employee(fst,lst) {
        hours=hrs;
        wage=wg;
}
void HrlyWorker::print() {
        Employee::print(); // Base class version
        cout<<getPay()<<endl;
}
double HrlyWorker::getPay() {
        return wage*hours;
}
```

```
main(){
    HrlyWorker h("Bob", "smith", 40.0, 10.0);
    h.print();
}
```

Constructor and Destructor in Derived classes

- If the derived class needs a constructor and base class has a constructor, then that constructor must be called.
- if that constructor needs arguments, then such arguments must be provided.
- It cannot directly initialize members of a base but via the constructor of base only
- Class members are constructed in the top-bottom order of the hierarchy. i.e. first the base class member, and then the derived class member
- They are destroyed in opposite order.

Example-1

```
class Point{
        protected: int xCo; int yCo;
        public:
            Point();
            ~Point();
};
class Circle:public Point{
        double radius;
        public:
            Circle();
            ~Circle();
};
Point::Point(): xCo(0),yCo(0){
        cout<< "Point constructor:"<<'['
        <<xCo<<','<<yCo<<']'<<endl;
}
Point::~Point(){
        cout<<"Point destructor:"<<'['
        <<xCo<<","<<yCo<<']'<<endl;
}
```

```
Circle::Circle(): radius(0.0) {
      cout<<"Circle constructor:rad:"
      <<radius<<endl<<'['<<xCo<<','
          <<yCo<<']'<<endl;
}
Circle::~Circle(){
    cout<<"Circle Destructor:rad:"
    <<radius<<endl<<'['<<xCo<<','
          <<yCo<<']'<<endl;
}
main(){
          Circle cl3;
}
```

Example-2

```
class Point{
        protected: int xCo; int yCo;
        public:
            Point(int, int);
            ~Point();
};
class Circle:public Point{
        double radius;
        public:
            Circle(double,int, int);
            ~Circle();
};
Point::Point(int a, int b) : xCo(a),yCo(b){
        cout<< "Point constructor:"<<'['
        <<xCo<<','<<yCo<<']'<<endl;
}
Point::~Point(){
        cout<<"Point destructor:"<<'['
        <<xCo<<","<<yCo<<']'<<endl;
}
```

```
Circle::Circle(double r, int a, int b)
          : Point(a,b),radius(r) {
     cout<<"Circle constructor:rad:"
     <<radius<<endl<<'['<<xCo<<','
          <<yCo<<']'<<endl;
}
Circle::~Circle(){
    cout<<"Circle Destructor:rad:"
    <<radius<<endl<<'['<<xCo<<','
          <<yCo<<']'<<endl;
}
main(){
          { Point p(11,22); }
          Circle cl1(4.5,72,29);
          Circle cl2(10.0,86,92);
          Circle cl3(15.0,43,43);
}
```

Example-3

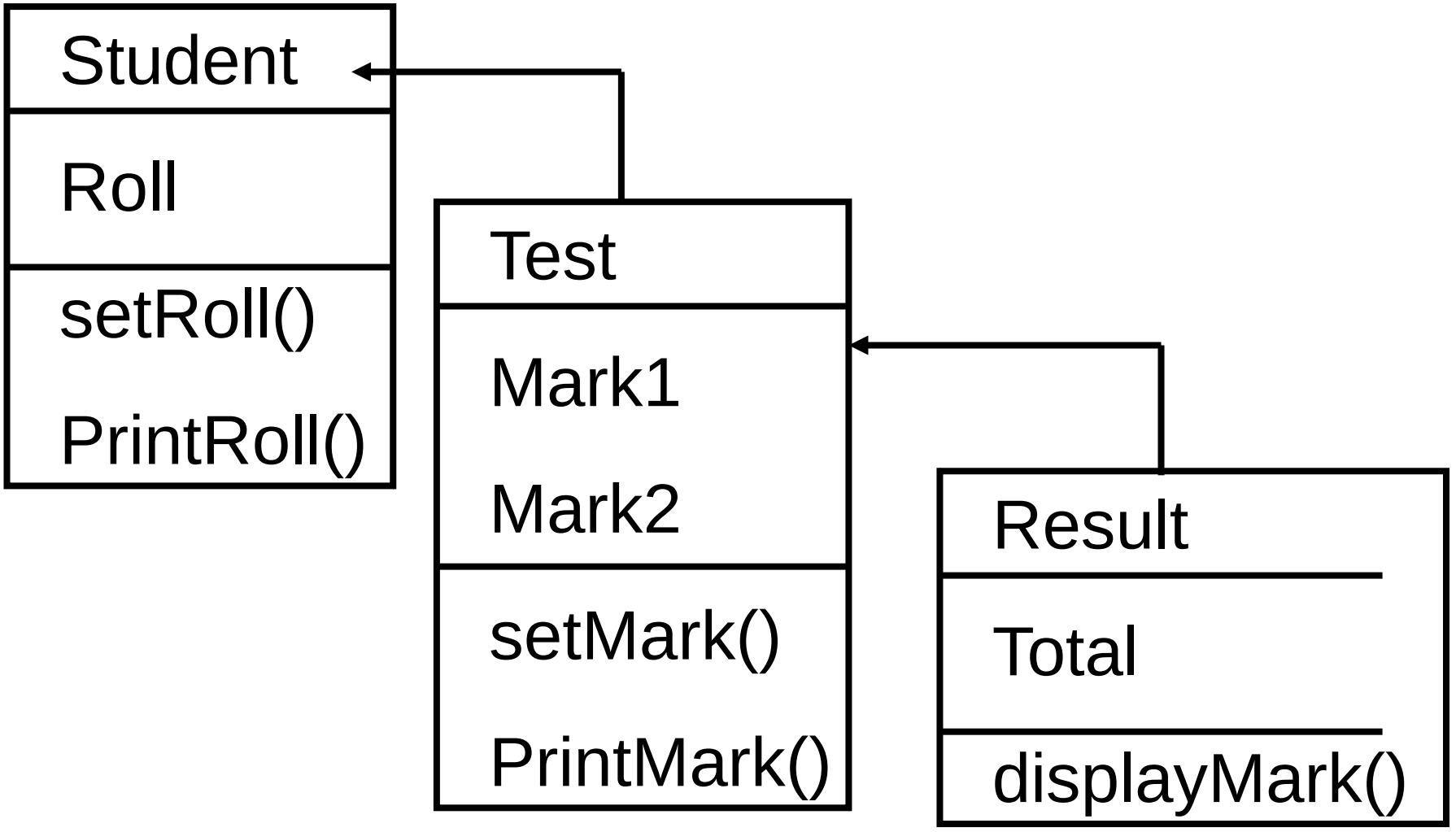
```
class mother {
  public:
    mother()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};
class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\
n"; }
};
class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};
```

```
int main () {
  daughter cynthia(0);
  son daniel(0);

  return 0;
}
```

Output:

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Multi-level inheritance



Example

```
class Student{
    protected:
        int roll;
    public:
        void setRoll(int a){roll=a;}
        void printRoll(void){
            cout<<“Roll:”<<roll<<endl;
        }
};
class Test:public Student{
    protected:
        float mark1;
        float mark2;
    public:
        void setMark(float a, float b) {
            mark1=a; mark2=b;
        }
    void printMark(){ printRoll();
            cout<<“Sub1:”<<mark1<<endl;
            cout<<“Sub2:”<<mark2<<endl;
    }
};
```

```
class Result:public Test {
    public:
      void displayMark(){
          printMark();
          total=mark1+mark2;
      }
    private:
          float total;
};
main(){
      Result s1;
      s1.setRoll(102);
      s1.setMark(75.0,30.0);
      s1.displayMark();
}
```

Example(2)

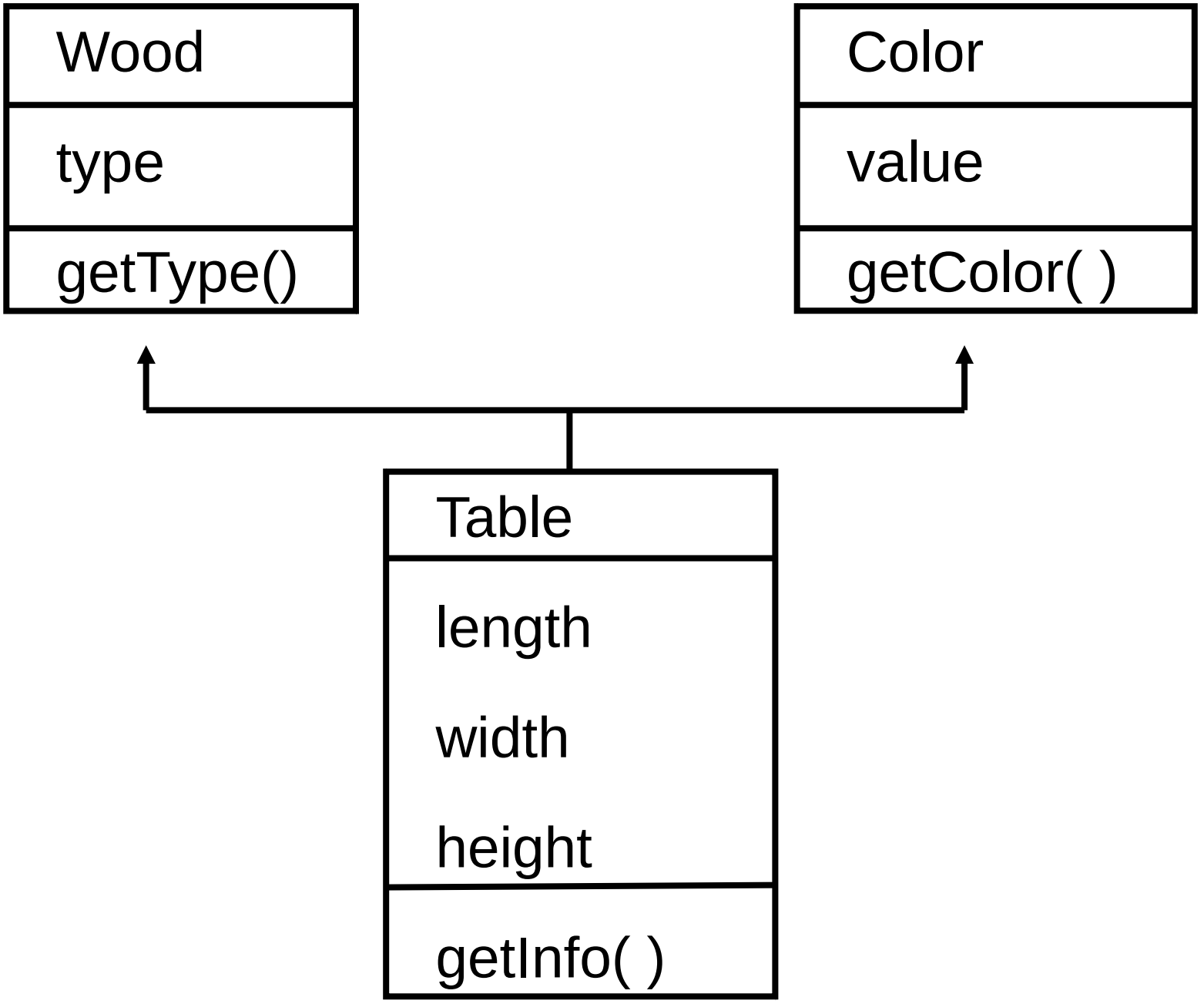
```
class Student{
    protected:
        int roll;
    public:
        void setRoll(int a){roll=a;}
        void printRoll(void){
            cout<<“Roll:”<<roll<<endl;
        }
};
class Test:public Student{
    protected:
        float mark1;
        float mark2;
    public:
        void setMark(float a, float b) {
            mark1=a; mark2=b;
        }
    void printMark(){ printRoll();
            cout<<“Sub1:”<<mark1<<endl;
            cout<<“Sub2:”<<mark2<<endl;
    }
};
```

```
class Result : public Test {
    public:
      void displayMark(){
        cout<<“ Roll:”<<roll;
        cout<<“ Sub1:”<<mark1;
        cout<<“ Sub2:”<<mark2;
        total = mark1+mark2;
        cout<<“ Total:”<<total;
      }
    private:
        float total;
};
main(){
    Result s1;
    s1.setRoll(102);
    s1.setMark(75.0,30.0);
    s1.displayMark();
}
```

Multiple inheritance

- When a class is derived from more than one class, it is called the multiple inheritance
- It means that a derived class inherits the members of several base classes.
- It is used when Type A is a kind of Type B and Type C and …
- The properties and behaviors of all the parents (base classes) are inherited to the child (derived class).

An example

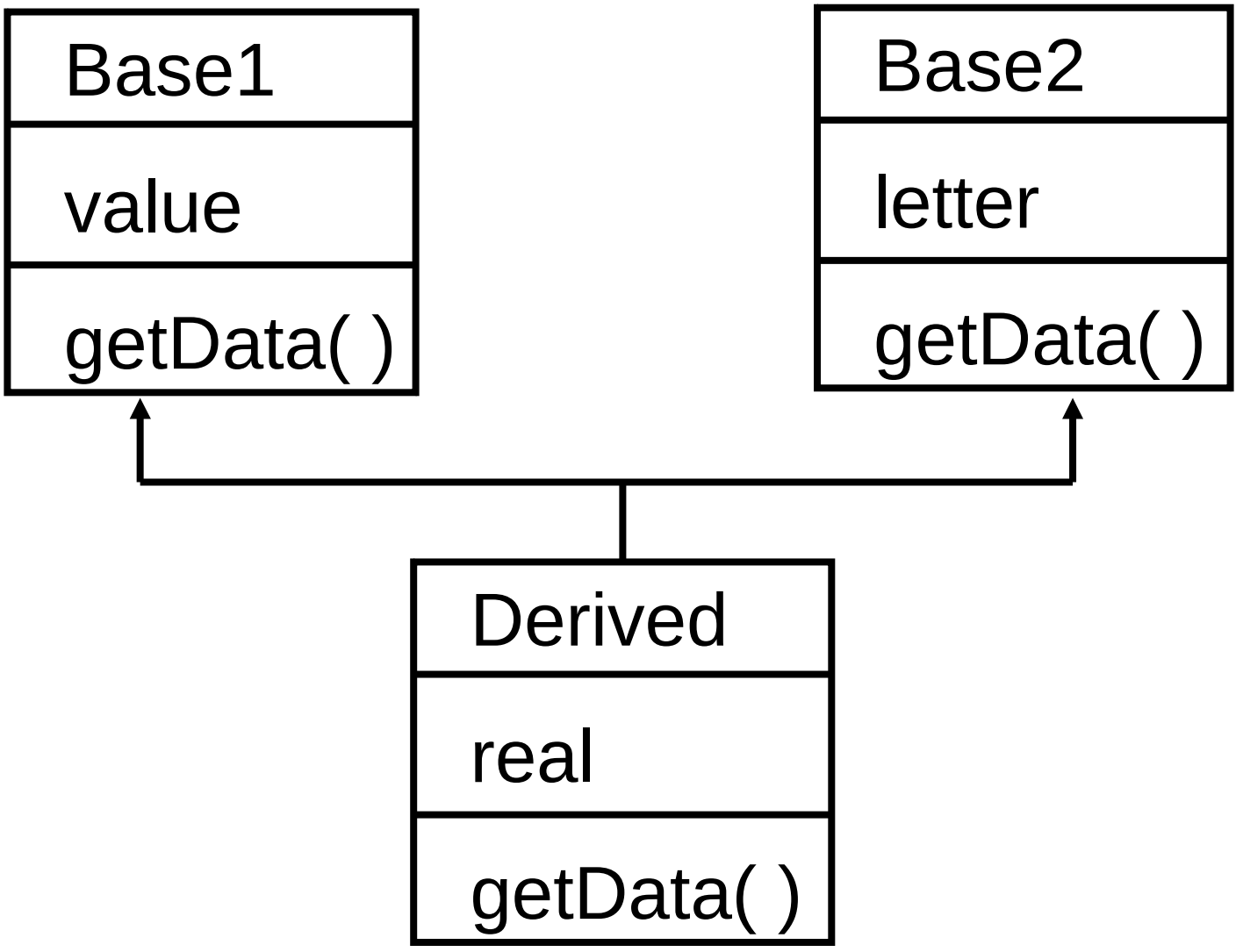

Wood
type
getType()
Color
value
getColor( )
Table
length
width
height
getInfo( )

An Example(2)

```
class Wood{
    public:
            Wood(char x):type(x) { }
            char getType() {
                return type;
            }
      protected:
            char type;
};
class Color{
    public:
      Color(char *c) {
       value = new char[strlen(c)+1];
          strcpy(value,c);
       }
      ~Color(){ delete value; }
      char* getColor() {
            return value;
      }
    protected:
          char *value;
};
```

```
class Table:public Wood,public Color{
    public:
     Table(char t,char *c, float l, float h, float w)
 : Wood(t), Color(c), length(l),
     height(h), width(w) { }

        void printData() {
           cout << " length: " << length;
           cout << " width: " << width;
           cout << " height: " << height;
        }
    private:
        double length, height, width;
};
main(){
        char col[10]="blue";
        Table tb('A',col,4.0,3.0,3.0);
        cout<<tb.getType();
        cout<<tb.getColor();
        tb.printData();
}
```

Another example



Another example(2)

```
class Base1{
    public:
            Base1(int x):value(x) { }
            int getData() {
                return value;
            }
     protected:
            int value;
};
class Base2{
    public:
            Base2(char c):letter(c) { }
            char getData() {
                    return letter;
            }
    Protected:
            char letter;
};
```

```
class Derived:public Base1,public Base2{
    public:
        Derived(int i,char c, double f)
        : Base1(i),Base2(c),real(f) { }
        double getData() {
            return real;
        }
    private:
        double real;
};
main(){
        Base1 *b1Ptr = new Base1(10);
        Base2 *b2Ptr = new Base2(20);
        Derived *dPtr = new Derived(7,'A',3.5);
        cout << b1Ptr->getData();
        cout << b2Ptr->getData();
        cout << dPtr->getData();
        ...
        }
```

Another example(3)

```
class Base1{
    public:
            Base1(int x):value(x) { }
            int getData() {
                return value;
            }
     protected:
            int value;
};
class Base2{
    public:
            Base2(char c):letter(c) { }
            char getData() {
                    return letter;
            }
    Protected:
            char letter;
};
```

```
class Derived:public Base1,public Base2{
    public:
        Derived(int i,char c, double f)
        : Base1(i),Base2(c),real(f) { }
        double getData() {
            return real;
        }
    private:
        double real;
};
main(){
        Base1 b1(10), *b1Ptr;
        Base2 b2('c'), *b2Ptr;
        Derived d(7,'A',3.5);
        cout<<d.getData();
        b1Ptr=&d;
        cout<<b1Ptr->getData(); //??
        b2Ptr=&d;
        cout<<b2Ptr->getData(); //??
}
```

Object slicing

- An object of a publicly derived class can also be treated as an object of its base class.
- But using a base class object as a derived class object may cause an error.
- It is always valid to assign a derived class pointer to a base class pointer because derived class object is also a base class object but…
- The base class pointer can only visualize the base class part of the derived class object
- This is also called "**object slicing**" i.e. only the base part (small) of the derived object (bigger) is visible and another part is not visible to the base pointer

Object slicing(2)

- In other words the derived object (a big object) is sliced to become a base object (a small object).
- The Compiler in above case performs an implicit conversion of the derived class pointer to a base class pointer
- It is also allowed to assign a base class pointer to a derived class pointer but..
- Assigning a base class pointer directly to a derived class pointer is an inherently dangerous assignment!!!
- In this case compiler does not perform an implicit conversion.

Example: Object Slicing

```
class Point{
    protected:
        int xCo;
        int yCo;
    public:
        Point():xCo(0),yCo(0) {}
        Point(int a,int b): xCo(a),yCo(b) {}
        friend ostream& operator << (ostream&,Point&);
};
class Circle:public Point{
   protected:
        double radius;
   public:
        Circle(double r, int a, int b):  Point(a,b),radius(r) {}
        void area();
        friend ostream& operator << (ostream&,Circle&);
};
```

Example: Object Slicing(2)

```
int main(){
    Point *pPtr=NULL, p(30,50);
    Circle *cPtr=NULL, c(2.7,10,5);
    cout<<p;
    cout<<c;
    /* these are ok */

    pPtr=&c;
    cout<<*pPtr;
    /* Treats circle as a point, it can see the base part only */
   cPtr=(Circle *)pPtr;
   cout<<*cPtr;
   cout<<cPtr->area();
   /* casts base class pointer to derived class pointer ok*/


   pPtr=&p;
   cPtr=(Circle *)pPtr;
   cout<<*cPtr;
   cout<<cPtr->area();
   /* Dangerous: Treating Point as a circle */
}
```

Polymorphism

- The ability for objects of different classes related by inheritance to respond differently to the same message.
- Polymorphism is implemented in C++ via inheritance and virtual functions.
- This is also called as dynamic polymorphism. i.e. when a request is made through a base class pointer to use a virtual function, C++ chooses the correct overridden function in the appropriate derived class associated with the object.

Another Example

```
class Base{
        ...
    public:
        void show(void){cout<<“Base”;}
};
class Derv1:public Base{
        ...
    public:
        void show(void){cout<<“Derv1”;}
};
class Derv2:public Base{
        ...
    public:
        void show(void){cout<<“Derv2”;}
};
```

```
main(){
        Base b;
        Base *ptr;
        Derv1 dv1;
        Derv2 dv2;
        ptr=&b;
        ptr->show();      //Base1
        ptr=&dv1;
        ptr->show();      //Base1
        ptr=&dv2;
        ptr->show();      //Base1
        }
```

- The compiler ignores the contents of the pointer ptr and chooses the member function that matches the type pointer.

Virtual function

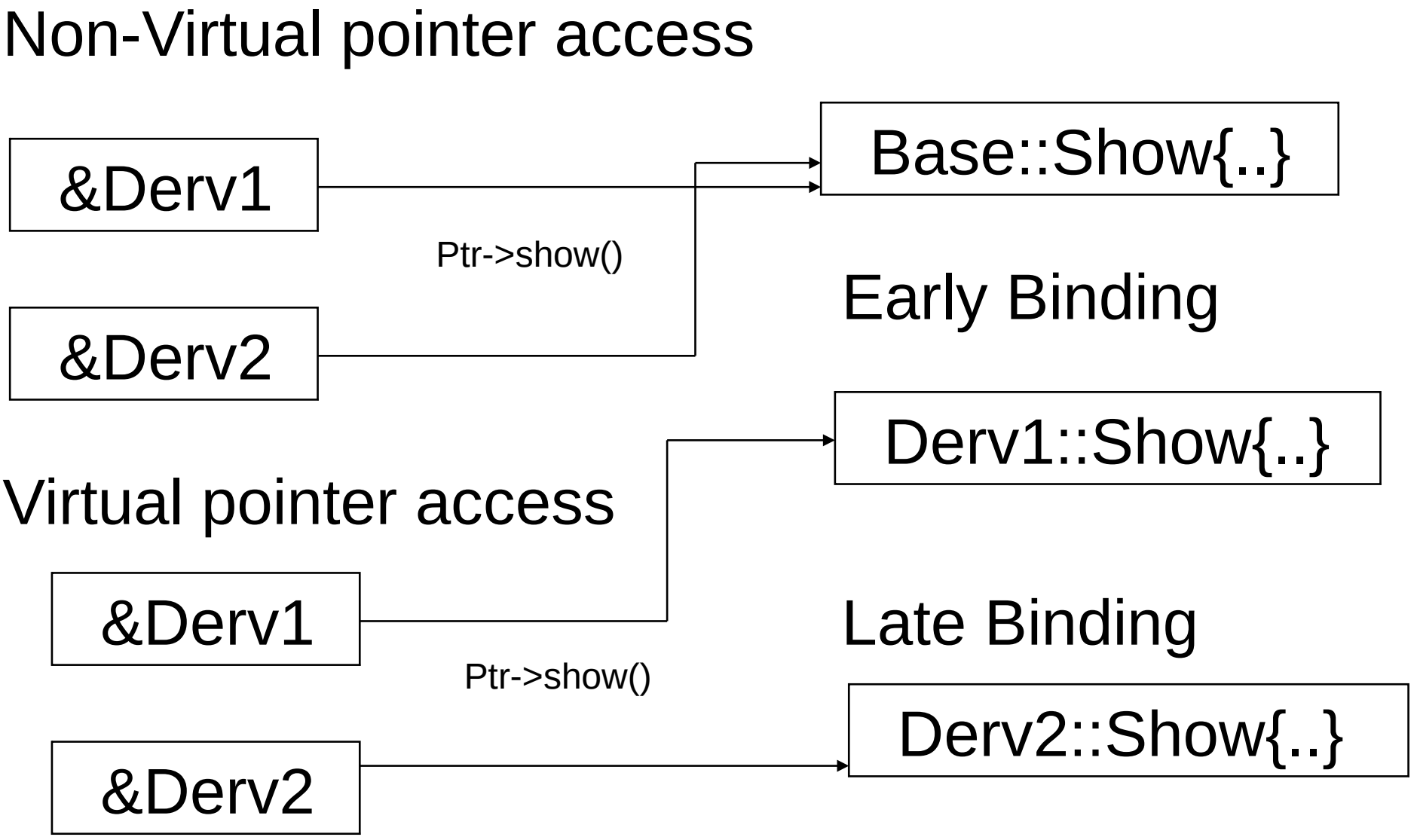
- If a function is declared as “virtual” in the base class then the implementation will not be decided at compile time but at the run time. This principle is called the late binding
- Once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy from that point, even if it is not declared virtual explicitly when a derived class overrides it
- But for the program clarity declare these functions as virtual explicitly at every level of hierarchy

example

- Modify Base class of the previous program as follows

```
class Base{
    public:
      virtual void show(){
          cout<<“Base”;
      }
};
```

Virtual Functions (contd.)



Virtual function Example

```
class Base{
        ...
    public:
        virtual void show(void)
    {cout<<“Base”;}
};
class Derv1:public Base{
        ...
    public:
        void show(void){cout<<“Derv1”;}
};
class Derv2:public Base{
        ...
    public:
        void show(void){cout<<“Derv2”;}
};
```

```
main(){
        Base b;
        Base *ptr;
        Derv1 dv1;
        Derv2 dv2;
        ptr=&b;
        ptr->show();      //Base1
        ptr=&dv1;
        ptr->show();      //Derv1
        ptr=&dv2;
        ptr->show();      //Derv2
        }
```

Purely virtual function

- If there can be no meaningful definition of a virtual function within a base class and we want to ensure that all derived classes override a virtual function.
  - Solution is to use a purely virtual function.
- It is a virtual function having no body
  - syntax: virtual void show()=0;
- Pure virtual functions may be inherited/overridden in derived classes.
- If the derived class fails to override, it may result in a compile-time error .

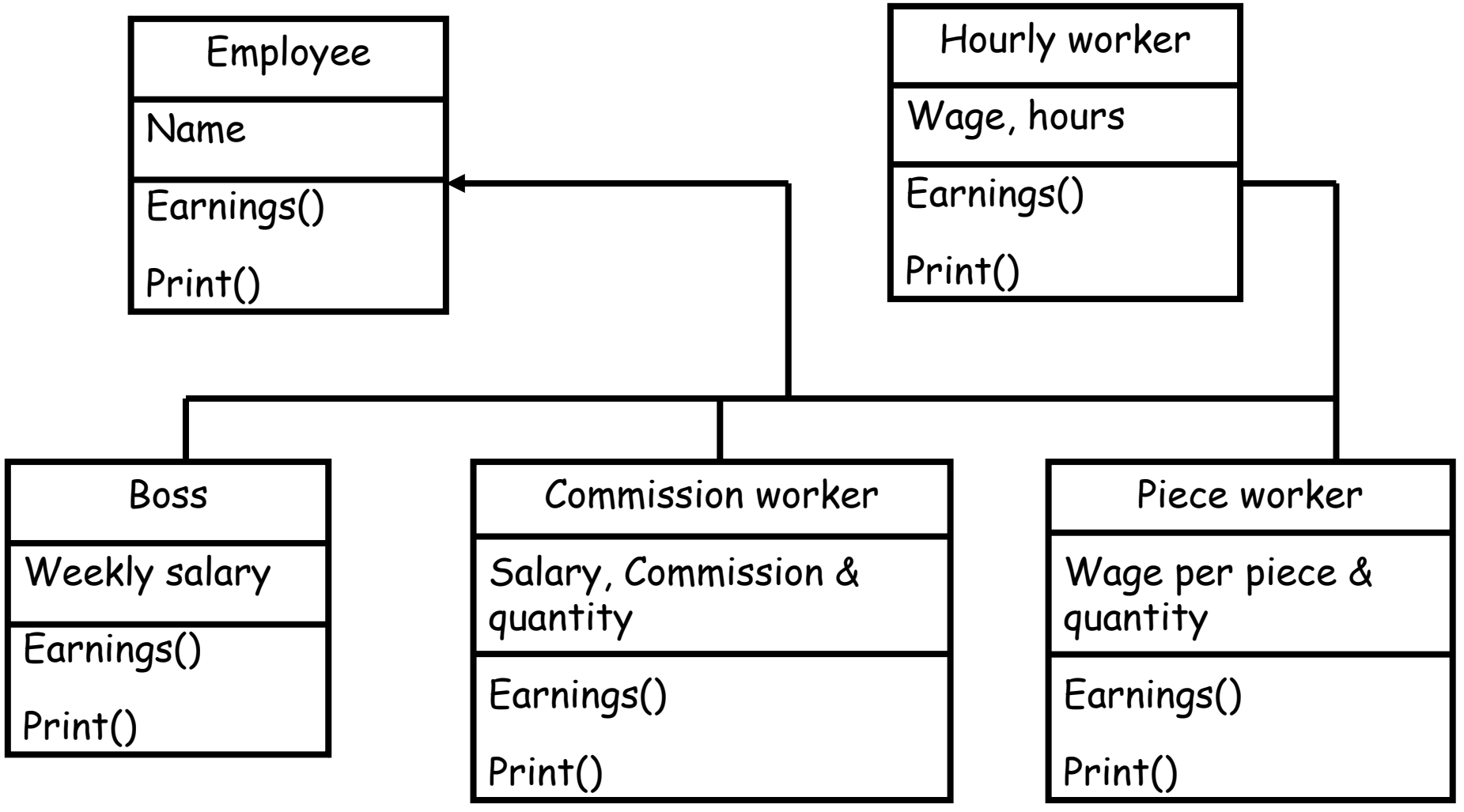
Abstract class vs Concrete class

- A class never used for instantiation is called an abstract class.
- The sole purpose of an abstract class is to provide an appropriate base class for derivation of sub classes.
- A class will be treated as an abstract base class if at least one member function of the class is purely virtual
- Attempting to instantiate an object of an abstract is a syntax error

Abstract classes

- If a class is derived from a class with a pure virtual function and if no definition for that function is supplied in the derived class also then …
  - That virtual function also remains purely virtual in the derived class.
  - The consequence is that the derived class again becomes an abstract class
  - i.e. It is not possible now to instantiate the derived class also!!!

Example


Employee
Name
Earnings()
Print()
Hourly worker
Wage, hours
Earnings()
Print()
Boss
Weekly salary
Earnings()
Print()
Commission worker
Salary, Commission & quantity
Earnings()
Print()
Piece worker
Wage per piece & quantity
Earnings()
Print()

Example(2)

```
class Employee{
    public:
    Employee(char *);
    ~Employee();
          virtual double earnings()=0;
    virtual void print();
    protected:
        char *name;
};
```

```
class Boss:public Employee{
    public:
      Boss(char*,double);
      virtual double earnings();
      virtual void print();
    private:
        double weeklySalary;
    };
```

Example(3)

```
class CommissionWorker:public Employee{
    public:
        CommissionWorker(char *, double,double,int);
        virtual double earnings();
        virtual void print();
    private:
        double salary;
        double commission;
        int quantity;
};
```

```
class PieceWorker:public Employee {
    public:
            PieceWorker(char *, double, int);
            virtual double earnings();
            virtual void print();
    private:
            double wagePerPiece;
            int quantity;
};
```

Example(4)

```
class HrlyWorker:public Employee{
  public:
HrlyWorker(char *, double,double);
    virtual double earnings();
    virtual void print();
  private:
       double wage;
       double hrs;
    };
```

```
void viaPointer(Employee *base){
        base->print();
    cout<<"Earned Rs."<<base->earnings();
}
void viaReference(Employee &base){
        base.print();
    cout<<"Earned Rs."<<base.earnings();
}
```

Example(5)

```
int main(){
        Boss b("John",800.0);
        b.print();  //static binding
        cout<<"Earned Rs."<<b.earnings();
        viaPointer(&b);
        viaReference(b);
        CommissionWorker c("Lee",200.0,3.0,15);
        viaPointer(&c);
        PieceWorker p("Margaret",2.5,200);
        viaReference(p);
        HrlyWorker h("Sue",2.5,200);
       h.print();
       cout<<"Earned Rs."<<h.earnings();
      viaPointer(&h);
      viaReference(h);
}
```

Observation

- At run time, when it is known what type of object is pointed to by base, then the appropriate version of the function will be called.
- The same message passed takes on many forms at different times.

Polymorphism via abstract class

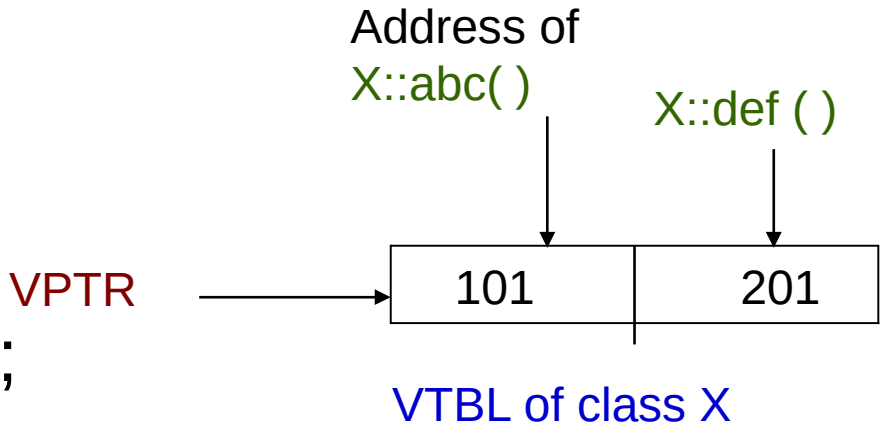
- An abstract class defines an interface for the various members of a class hierarchy.
- The abstract class contains pure virtual function and the hierarchy can use this interface through polymorphism.
- Other types of polymorphism are obtained through function or operator overloading are called static polymorphism.
- This powerful feature of C++ helps in reducing complexity of the system.

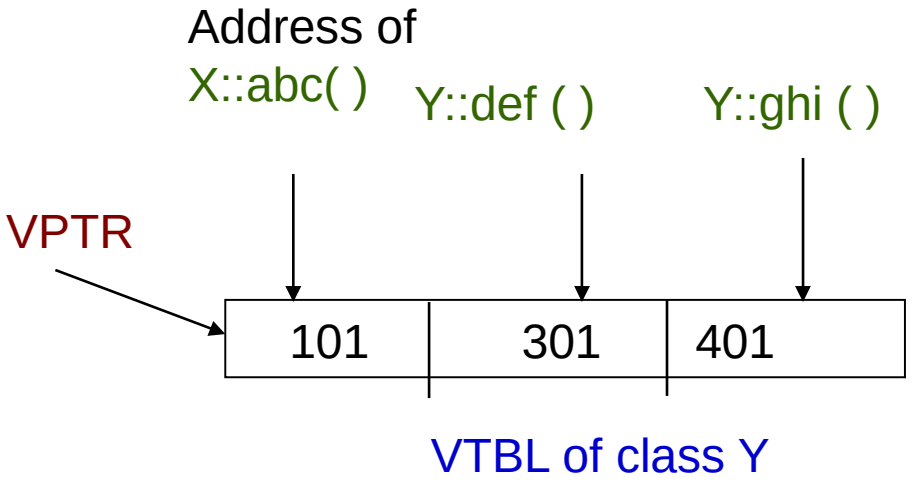
Mechanism of Virtual Functions

- For every base class that have one or more virtual functions, a table of function addresses is created during run time.
- This table is called virtual table or VTBL and contains the address of virtual functions only.
- Each object of such classes contains a pointer to VTBL of corresponding class called VPTR
- Whenever a call is dispatched to a virtual function through an *object* or *reference* or *pointer* to the object, then value of VPTR of that object is read, then the address of called function from the VTBL is obtained
- The size of the objects of the classes containing virtual function increases by a size of a pointer (for VPTR)

Virtual Table (VTBL)

```
class X{
   public:
        virtual void abc();
        virtual void def();
};
class Y:public X{
   public:
        virtual void def();
        virtual void ghi();
   };
```





A problem with destructors

- Base *ptr=new Derived; delete ptr;
- In case of polymorphism, If an object is destroyed explicitly by applying the delete operator to a base class pointer, then always the base class destructor called on the object.
- The result is the base portion of the object is only destroyed. But the derived class object should be destroyed first and then the base object.
- Therefore, declare virtual to base class destructor, which makes all derived class destructors virtual even though they do not have the same name as the the base destructor.

Virtual destructor

- C++ provides a default destructor for your classes if you do not provide one yourself
- When we want to provide our own destructor (particularly if needs to deallocate memory). You should always make your destructors virtual if you’re dealing with inheritance

An example

```
class Base {
public:
    ~Base() {
        cout << "Calling ~Base()" << endl;
    }
};
class Derived: public Base {
private:
    int* m_pnArray;
public:
    Derived(int nLength) {
        m_pnArray = new int[nLength];
    }
     ~Derived() // note: not virtual  {
            cout << "Calling ~Derived()" << endl;
            delete[] m_pnArray;
    }
};
```

```
int main() {
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;

    return 0;
}
```

Output:

Calling ~Base()

An example(2)

```
class Base {
public:
    virtual ~Base() {
      cout << "Calling ~Base()" << endl;
    }
};
 class Derived: public Base {
private:
    int* m_pnArray;
 public:
    Derived(int nLength) {
     m_pnArray = new int[nLength];
    }
    virtual ~Derived() {
        cout << "Calling ~Derived()" << endl;
        delete[] m_pnArray;
    }
};
```

```
int main() {
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;

    return 0;
}
```

Output:

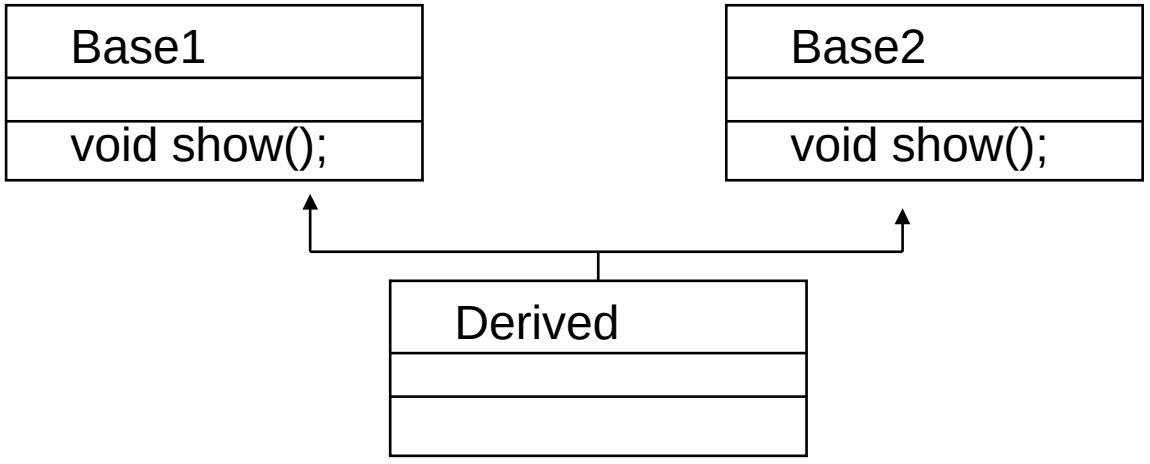
Calling ~Derived()
Calling ~Base()

Virtual Constructors !

- Base *ptr = new Derived;
- If the constructor is virtual then the above statement will call the derived class constructor only and the Base part of the object could not be created.
- Therefore declaring constructors as virtual in not allowed.

Ambiguity in inheritance

- Identical Members in more than one Base class
- In case of multiple inheritance, if two or more base classes have same member and the derive class does not overrides it.
- Then compiler will not be able to decide which of the two functions to call, in case the derive class object is trying to access the same function of the base classes

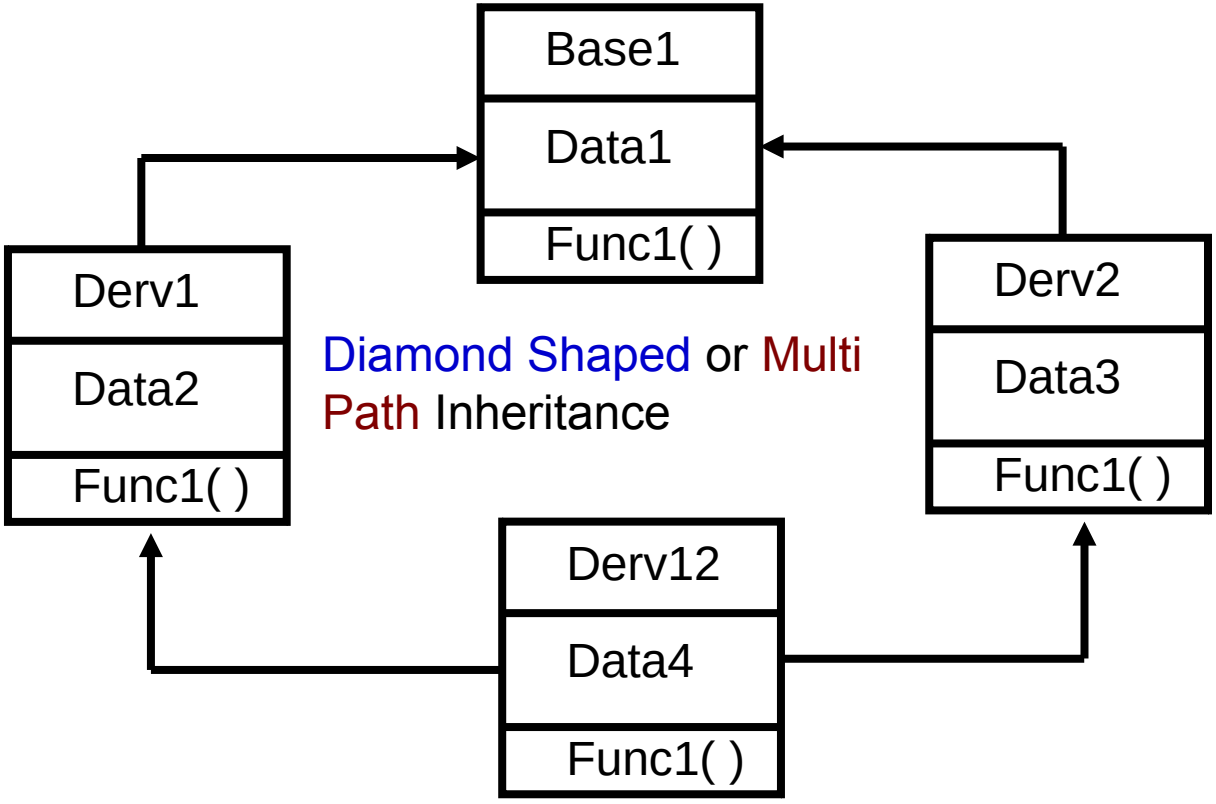


Derived d;

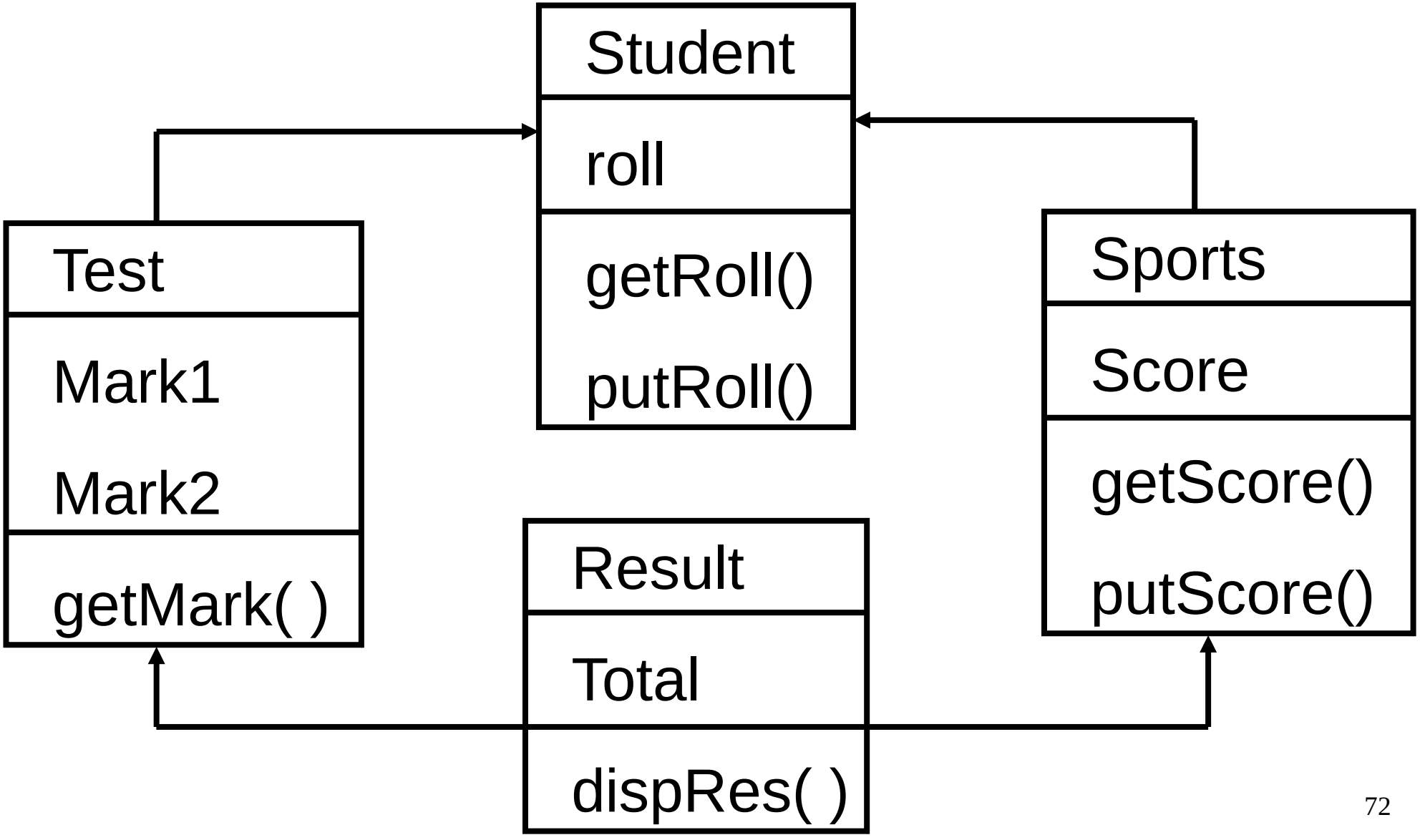
d.show() //which one to call?

Ambiguity in inheritance

- If two or more classes in turn inherit from a common base class?
- In the example, the derv12 class gets two copies of Base1
- One copy Via Derv1 and another copy via Derv2.
- Which is an ambiguous situation



Example


Student
roll
getRoll()
putRoll()
Test
Mark1
Mark2
getMark( )
Sports
Score
getScore()
putScore()
Result
Total
dispRes( )

Example

```
class Student{
    protected:
        int roll;
    public:
        void getRoll(int);
        void putRoll();
    };

class Test:public Student{
    protected: float mark1,mark2;
    public: void getMark(float,float);
 };
```

```
class Sports: public Student{
    protected:
        int score;
    public:
        void getScore(int);
        void putScore();
    };

class Result:public Test,public Sports{
    protected: float total;
    public: void dispRes(void);
 };
```

Example(2)

```
main(){
    Test one;
    Sports two;
    Result both;
    Student *array[3];
    array[0]=&both;
    array[1]=&one;
    array[2]=&two;
    array[0]->getRoll(101);
    array[0]->putRoll();

}
```

Virtual base class

- It is ambiguous to declare an object of Result type as it will try to create two copies of Student type.
- To avoid above type of ambiguity we have to inherit class student and sports as virtual
- Syntax
  - class Test: virtual public Student
  - class Sports: public virtual Student
- Then the Result object will have only one sub-object of Student type.

End of Inheritance & Polymorphism Slide