

Inheritance

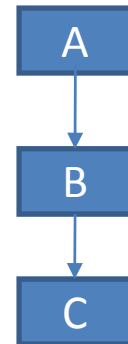
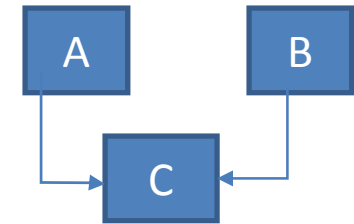
Introduction

- **Inheritance**

- New classes created from existing classes
- Extends *Reusability* of existing attributes and behaviors
- The existing class is called as ***Base*** class or ***Super*** class or ***Parent*** class.
- Derived class
 - Class that inherits data members and member functions from the existing class.
 - It is also called as ***Sub*** class or ***Child*** class.

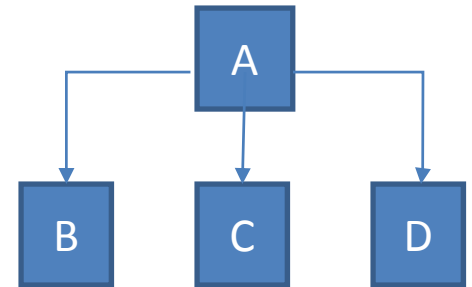
Types of Inheritance

- Inheritances are of 5 types
 - **Simple inheritance:** The child class derives from only one parent class.
 - **Multiple inheritance:** The child class derives from multiple parent classes.
 - **Multilevel inheritance:** The child class is derived from another derived class.

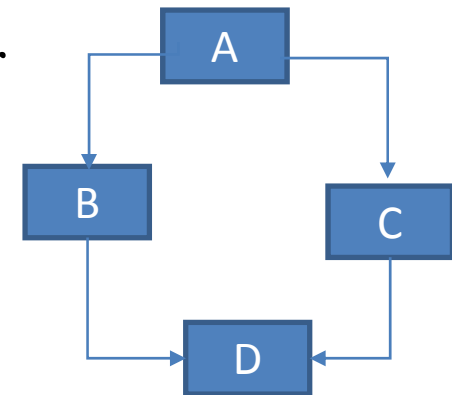


Types of Inheritance (Contd..)

- **Hierarchical inheritance:** The traits of one base class are derived by several child classes.

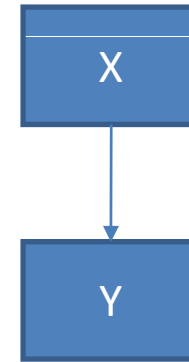


- **Hybrid inheritance:** Combination of two or more inheritances.



```
class X{  
};
```

```
class Y : visibility-mode X{  
};
```



- Visibility modes are of 3 types.
- public
 - protected
 - private

Public Visibility Mode

```
class Y : public X
{
// Class Y now inherits the members of Class X publicly
}
```

base class (X)

public members

protected members

private members



derived class (Y)

public

protected

Can't be inherited

Protected Visibility Mode

```
class Y : protected X
{
    // Class Y now inherits the members of Class X protectedly
}
```

base class (X)

public members

protected members

private members



derived class (Y)

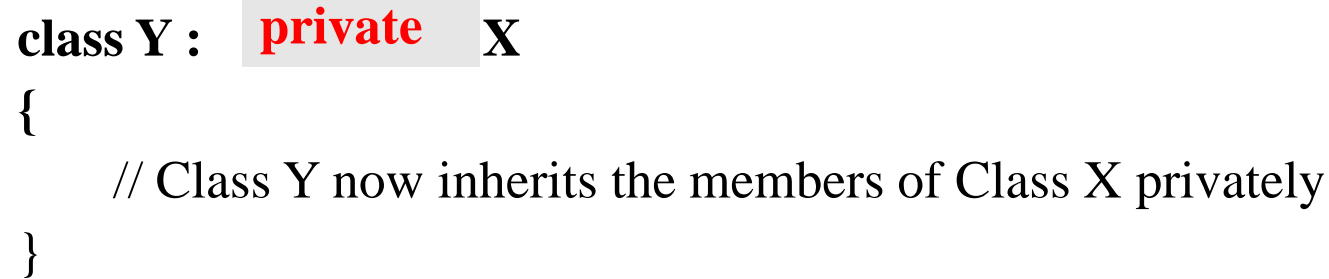
protected

protected

Can't be inherited

Private Visibility Mode

Not Compulsory



```
class Y : private X
{
    // Class Y now inherits the members of Class X privately
}
```

base class (X)

public members

protected members

private members

derived class (Y)

private

private

Can't be inherited

Difference Between Private and Protected

Private

- ✗ Data Members can be accessed only by member function of the same class.
- ✗ Data Members cannot be accessed outside the class

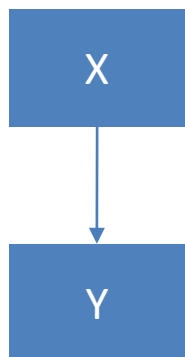
Protected

- ✗ Can be accessed by the member function of the same class
- ✗ Data Members can be accessed by the member function of the derived class.
However, in that class it is ***protected***?

Single Inheritance

```
class X{  
  int a;  
  public:  
    int b;  
    void get()  
    { a=5; b=10;}  
    void disp()  
    { cout<<a<<b; }  
};
```

```
class Y: public X{  
  int c;  
  public:  
    void set()  
    { c=15;}  
    void show()  
    { cout<<c; }  
};
```



```
main(){  
  Y y1;  
  y1.get();  
  y1.disp();      // 5, 10  
  y1.set();  
  y1.show();      // 15  
}
```

```
class Y{  
  int c;  
  public:  
    int b;  
    void get() {a=5; b=10;}  
    void disp() {cout<<a<<b;}  
    void set() {c=15;}  
    void show() {cout<<c;}  
};
```

Multiple Inheritance

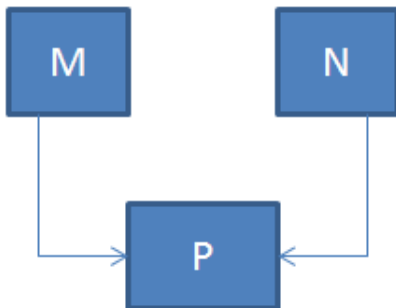
```
class M
{
protected:
int m;
public:
void get_m(int x)
    { m=x; }
};

class N
{
protected:
int n;
public:
void get_n(int y)
    { n=y; }
};
```

```
class P:public M,public N
{
public:
void display(void);
};

void P::display(void)
{
cout<<"m=" <<m <<"\n";
cout<<"n=" <<n <<"\n";
cout<<"m*n=" <<m *n<<"\n";
}
```

```
int main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();
return 0;
}
```



↓

```
class P{
Protected:
int m,n;
public:
void get_m(int x) {m=x;}
void get_n(int y) {n=y;}
void display(void)
{
    cout<<"m="<<m<<"\n";
    cout<<"n="<<n<<"\n";
    cout<<"m*n="<<m*n<<"\n";
}
};
```

OUTPUT:

```
m=10
n=20
m*n=200
```

Ambiguity in Multiple Inheritance

```
class X{  
public:  
void disp()  
    { cout<<"class X" ;  
      }  
};
```

```
class Y{  
public:  
void disp()  
    { cout<<"class Y" ;  
      }  
};
```

```
class Z:public X,public Y{  
public:  
void disp()  
    { cout<<"class Z" ;  
      }  
};
```

```
main(){  
    Z z1;  
    z1.disp();    //class Z  
}
```

Ambiguity resolution in Inheritance (Member Function Overriding)

```
class X{  
public:  
void disp()  
{ cout<<"class X" ; }  
};
```

```
class Y{  
public:  
void disp()  
{ cout<<"class Y" ; }  
};
```

```
class Z:public X,public Y  
{  
public:  
void disp()  
{ cout<<"class Z" ; }  
};
```

Method overriding, is a feature that allows a child class to provide a specific implementation of a method that is already provided by one of its parent classes. The implementation in the child class overrides (replaces) the implementation in the parent class by providing a method that has same name, same parameters, and same return type as the method in the parent class. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

```
main(){  
Z z1;  
z1.X::disp();           // class X  
z1.Y::disp();           // class Y  
z1.Z::disp();           // class Z  
z1.disp();              // class Z  
}
```

Multilevel inheritance

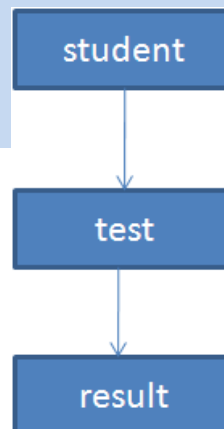
```
class student
{
protected:
int roll_no;
public:
void get_no(int a)
{ roll_no=a; }
void put_no(){
cout<<"Roll
number:"<<roll_no ;}
};
```

```
int main(){
result student1;
student1.get_no(111);
student1.get_marks(75.0,
59.5);
student1.display();
return 0;
}
```

```
class test:public student
{
protected:
float sub1; float sub2;
public:
void get_marks(float x,float y)
{ sub1=x; sub2=y; }

void put_marks(void){
cout<<"marks in
sub1="<<sub1<<"\n";
cout<<"marks in
sub2="<<sub2<<"\n";
}
};
```

```
class result:public test
{
float total;
public:
void display(void){
total=sub1+sub2;
put_no();
put_marks();
cout<<"total="<<total; }
};
```



OUTPUT:
Roll number: 111
Marks in sub1=75
Marks in sub2=59.5
Total=134.5

Hierarchical inheritance

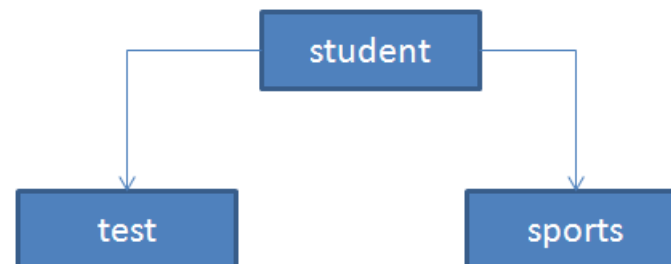
```
class student
{
protected:
int roll_no;
public:
void get_no(int a)
{ roll_no=a; }
void put_no(void){
    cout<<"roll_no:"<<
    roll_no; }
};
```

```
int main(){
test t;
sports s;
t.get_no(110);
t.get_marks(75.0,59.5);
s.get_no(220);
s.get_score(78);
t.put_no();           // 110
t.put_marks();        // 75, 59.5
s.put_no();           // 220
s.put_score();        // 78
}
```

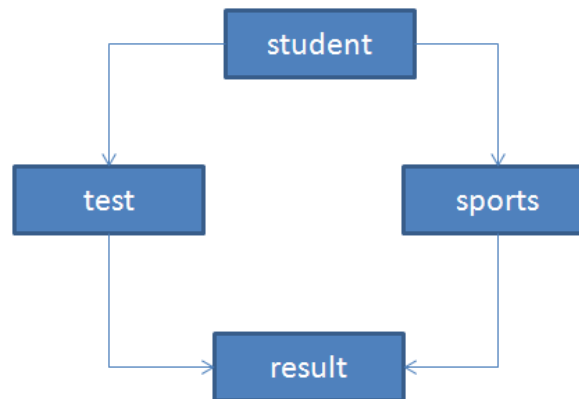
```
class test:public student
{
protected:
float sub1; float sub2;
public:
void get_marks(float x,float y)
{ sub1=x; sub2=y; }

void put_marks(void){
    cout<<"marks in
    sub1="<<sub1<<"\n";
    cout<<"marks in
    sub2="<<sub2<<"\n";
}
};
```

```
class sports:public student
{
protected:
float score;
public:
void get_score(float s)
{ score=s; }
void put_score(void)
{
    cout<<"sports
    wt:"<<score<<"\n";
} };
```



Hybrid inheritance



```
class student
{
protected:
int roll_no;
public:
void get_no(int a)
{ roll_no=a; }
void put_no(void){
    cout<<"roll_no:"<<
    roll_no; }
};
```

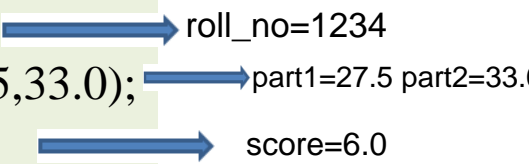
```
class test:public student
{
protected:
float part1,part2;
public:
void get_marks(float x,float y)
{ part1=x; part2=y; }
void put_marks(void) {
    cout<<"marks obtained:"<<"\n"
    <<"Part1="<<part1<<"\n"
    <<"Part2="<<part2<<"\n";
} };
```

```
class sports: public student
{
protected:
float score;
public:
void get_score(float s)
{ score=s; }
void put_score(void)
{
    cout<<"sports
    wt:"<<score<<"\n";
} };
```

```
class result:public test,public sports{
float total;
public:
void display(void){
    total=part1+part2+score;
    put_no();
    put_marks();
    put_score();
    cout<<"total score:"<<total<<"\n";
}
};
```


Hybrid inheritance (Contd..)

```
int main()
{
    result student1;
    student1.get_no(1234);
    student1.get_marks(27.5,33.0);
    student1.get_score(6.0);
    student1.display();
    return 0;
}
```

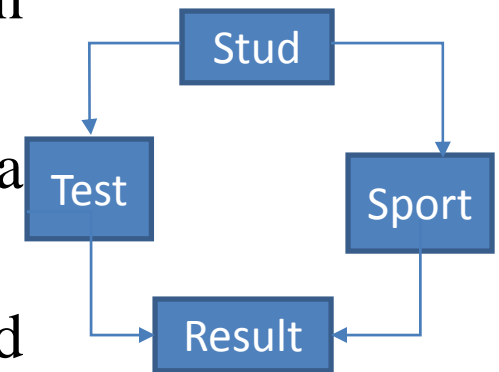


OUTPUT:

Roll_no:1234
Marks obtained:
Part1=27.5
Part2=33
Sports wt:6
Total score=66.5

Virtual base class

- Two copies of the traits of Stud will appear in Result class, that will lead to ambiguity
- The solution is to make the base class Stud as a virtual base class.
- This can be done by specifying virtual keyword while defining the **direct** derived classes.



```
class Test:public virtual Stud
{
};
```

```
class Sport:virtual public Stud
{
};
```

- virtual and public keywords may exchange their position.
- The Stud class has become virtual base class and only one copy of the traits from Stud will appear in Result.

Constructor in derived class

- As long as no base class constructor takes any arguments, the derived class need not have a constructor function.
- However, if any base class contains a constructor with one or more arguments, then it is mandatory for the class to have a constructor and pass the arguments to the base class constructors.
- Always the base constructor is executed first and then the constructor in the derived class is executed.
- Syntax for defining Derived Class Constructor:
<derived constructor>(<argument list>):base class constructor<argument list>

Methods of inheritance	Order of execution
Class B:public A { };	A(); base constructor B(); derived constructor
Class A:public B,public C { };	B(); base (first) C(); base (second) A(); derived
Class A:public B, virtual public C { };	C(); virtual base constructor B(); ordinary base constructor A(); derived

Constructor in derived class(example...)

```
class alpha{
int x;
public:
alpha(int i)
{
x=i;
cout<<"alpha initialized";
}
void show_x(void){
cout<<"x="<<x<<"\n";
}
};
```

```
class beta{
float y;
public:
beta(float j){
y=j;
cout<<"beta initialized";
}
void show_y(void){
cout<<"y="<<y<<"\n";
}
};
```

```
class gamma:public beta,public alpha
{
int m,n;
public:
gamma(int a,float b,int c,int
d):alpha(a),beta(b){
m=c;
n=d;
cout<<"gamma initialized";
}
void show_mn(void){
cout<<"m="<<m<<"\n"
<<"n="<<n<<"\n";
}
};
```

```
int main()
{
gamma g(5,10.75,20,30);
g.show_x();
g.show_y();
g.show_mn();
return 0;
}
```

Output:
beta intialized
alpha intialized
gamma intialized
x=5
y=10.75
m=20
n=30

Nesting(containership) of classes

- If an object of a class is becoming a member of another class, it is referred as **member object**.

```
class X { .....};           class Y { .....};           class Z {  
                                X x1;  
                                //x1 is an object of X class  
                                Y y1;  
                                //y1 is an object of Y class  
                                ..... };
```

In the above example, object of class Z contains the objects of class A & class B. This kind of relationship is called **containership or nesting**.

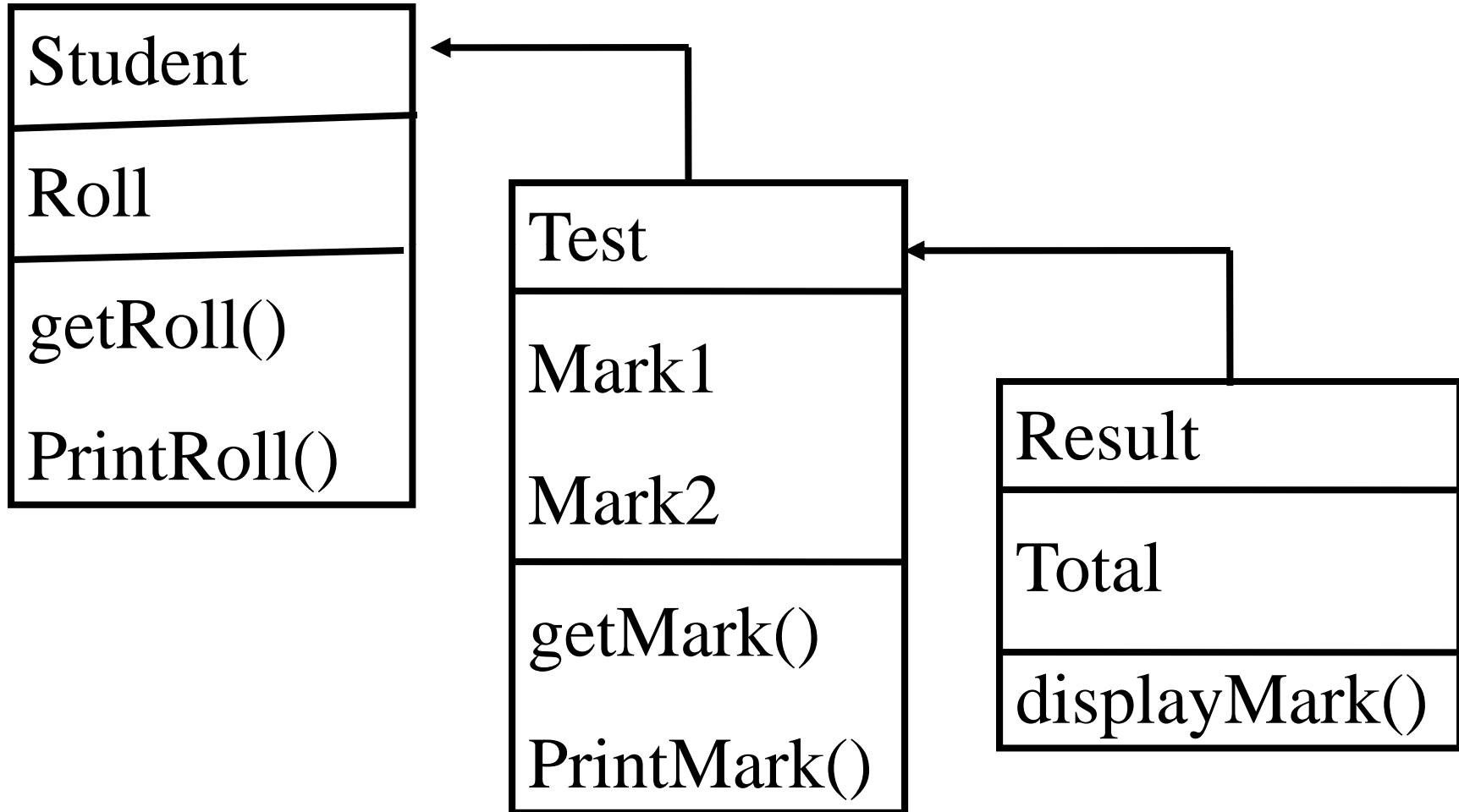
```
class A
{
public:
    void get()
    {
        cout<<"fun1";
    }
};
```

```
class B
{
public:
    void show()
    {
        cout<<"fun2";
    }
};
```

```
class C
{
    A obj1;
    B obj2;
public:
    void disp()
    {
        obj1.get();
        obj2.show();
    }
};
```

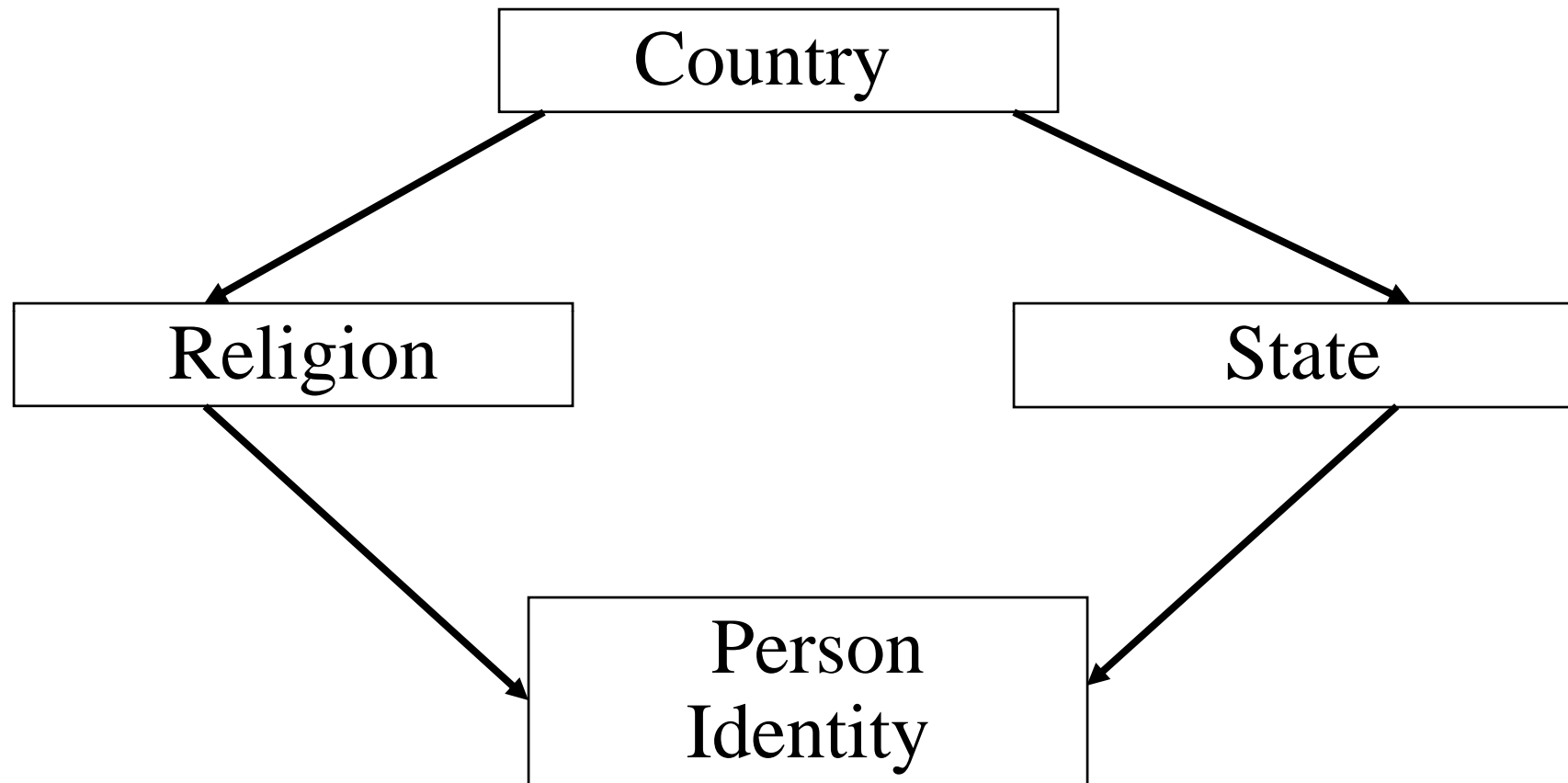
```
main(){
    C c1;
    c1.disp();
}
```

Assignment 1



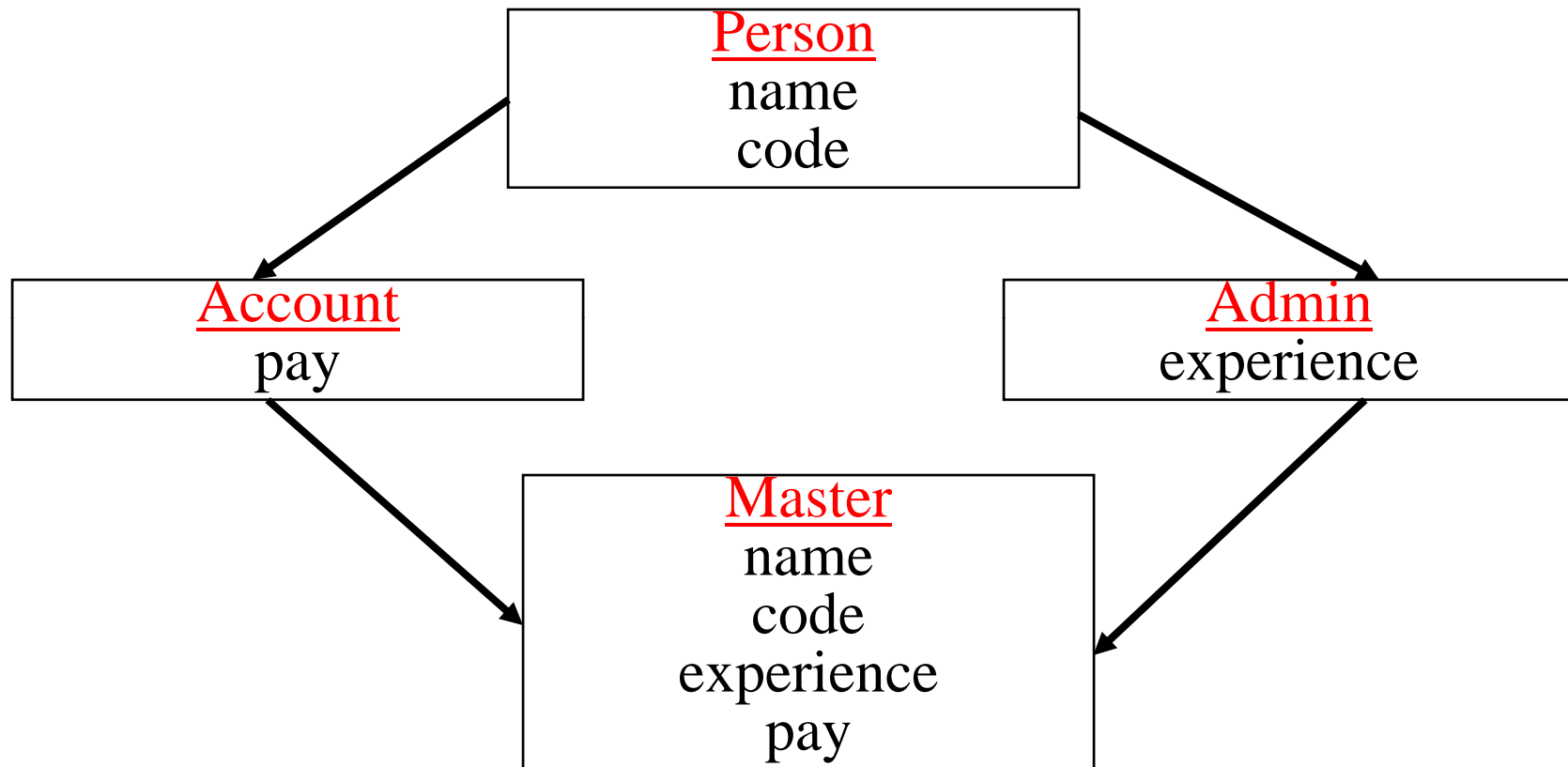
Output: Detail information about 10 students

Assignment 2



WAP to get the identity of a person ,where all the classes have suitable datamember & member function.

Assignment 3



WAP to make Person as virtual base class and create, update and display the information about the Master objects.

Note: Define the constructors for each class.