UNIT II
Inheritance and Pointers in C++

Contents

- **Inheritance-** Base Class and derived Class, protected members, relationship between base Class and derived Class, Constructor and destructor in Derived Class, Overriding Member Functions, Class Hierarchies, Public and Private Inheritance, Types of Inheritance, Ambiguity in Multiple Inheritance, Virtual Base Class, Abstract class, Friend Class , Nested Class.

- **Pointers:** declaring and initializing pointers, indirection Operators, Memory Management: new and delete, Pointers to Objects, this pointer, Pointers Vs Arrays, accessing Arrays using pointers, Arrays of Pointers, Function pointers, Pointers to Pointers, Pointers to Derived classes, Passing pointers to functions, Return pointers from functions, Null pointer, void pointer.

Introduction

- Inheritance
  - New classes created from existing classes
  - Absorb attributes and behaviors.
  - C++ supports the concept of reusability
- Derived class
  - Class that inherits data members and member functions from a previously defined base class

Introduction

- Inheritance
  - Single Inheritance
    - Class inherits from one base class
  - Multiple Inheritance
    - Class inherits from multiple base classes
  - Three types of inheritance:
    - **public:** Derived objects are accessible by the base class objects
    - **private:** Derived objects are inaccessible by the base class
    - **protected:** Derived classes and friends can access protected members of the base class

Inheritance in C++

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**
- Inheritance is one of the most important feature of Object Oriented Programming

  Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

  Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Implementing inheritance in C++

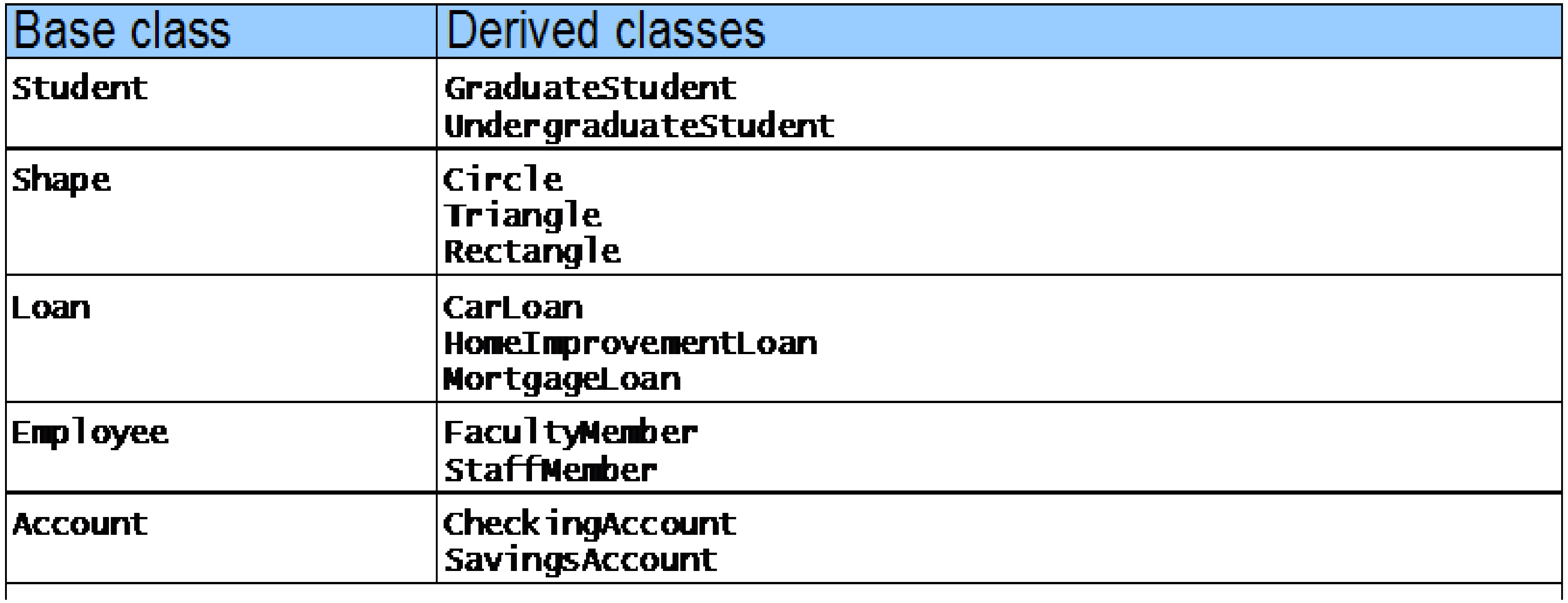
- For creating a sub-class which is inherited from the base class we have to follow the below syntax:

```
class subclass_name : access_mode base_class_name
{
 //body of subclass
};
```

Note: A derived class doesn’t inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares

Base and Derived Classes

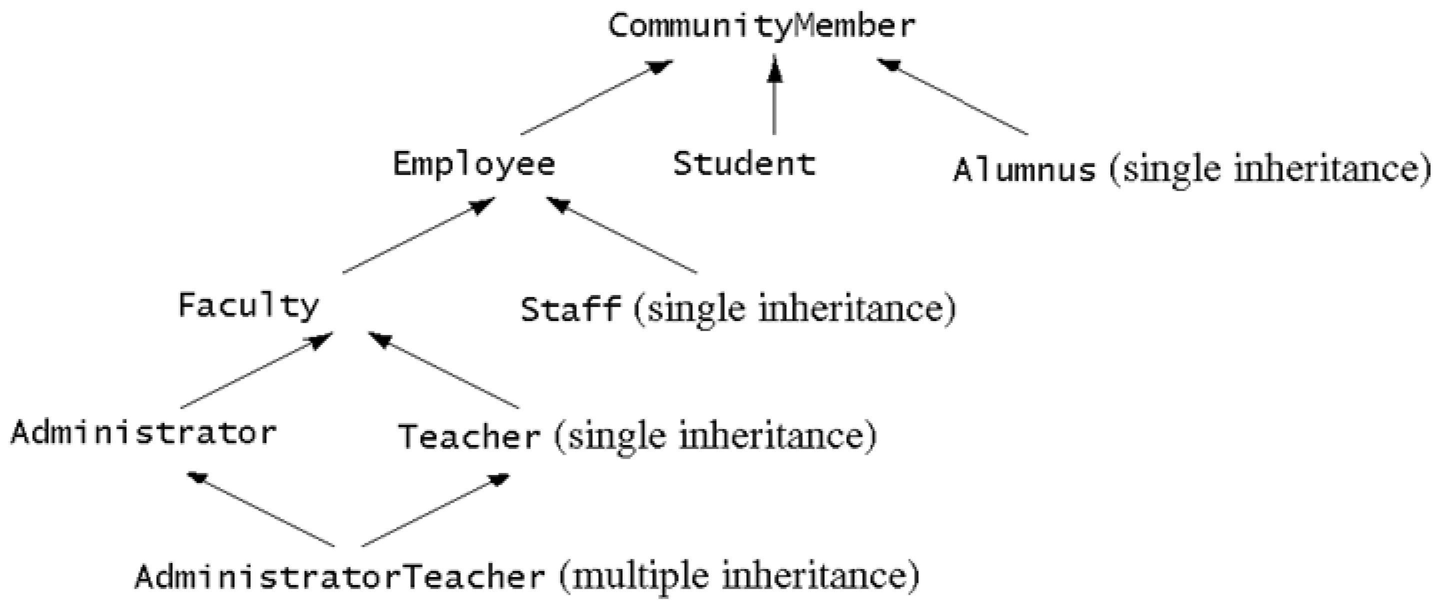
- Often an object from a derived class (subclass) “is an” object of a base class (superclass)
- Examples of inheritance



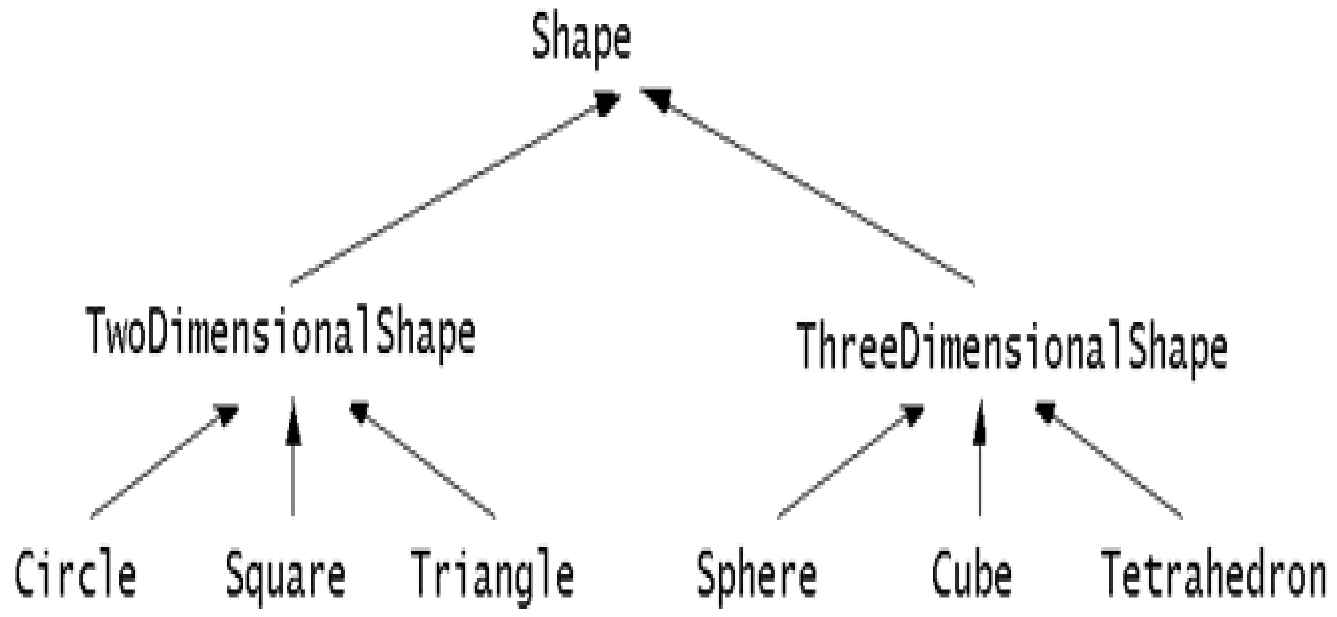
Base class	Derived classes
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan
Employee	FacultyMember StaffMember
Account	CheckingAccount SavingsAccount

Base and Derived Classes

- Inheritance Hierarchy for University CommunityMembers


CommunityMember
Employee
Student
Alumnus (single inheritance)
Faculty
Staff (single inheritance)
Administrator
Teacher (single inheritance)
AdministratorTeacher (multiple inheritance)

Base and Derived Classes



Base and Derived Classes

- Implementation of public inheritance

```
class CommissionWorker : public Employee {
 ...
};
```

Class CommissionWorker inherits from class Employee

- friend functions not inherited
- private members of base class not accessible from derived class

Protected Members

- **protected inheritance**
  - Intermediate level of protection between public and private inheritance
  - Derived-class members can refer to public and protected members of the base class simply by using the member names
  - Note that protected data “breaks” encapsulation

Program

```
class parent{
protected:
   int c;
public:
   void display(){
      c=100;
      cout<<"Base class"<<c;
   }
};
class child: public parent{
   public:
      void disp(){
           c=200;
           cout<<"Derived class"<<c;}
};
int main(){
  child obj;
  obj.disp();
  obj.display();
  obj.c=100;
  return 0;
}
```

Constructor and Destructors in Derived Classes

- Derived classes can have their own constructor and destructor
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Execution of base class constructor

Method of Inheritance	Order of execution

class Derived: public Base
{
};
	Base();
	Derived();

class C: public A, public B
{
};
	A(); //base(first)
	B(); //base(Second)
	C();//Derived

Example

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

```
/*output*/

Constructing base
Constructing derived
Destructing derived
Destructing base
```

The same general rule applies in situations involving multiple base classes

```
class base1 {
      public: base1() {
                cout << "Constructing base1\n";
      }
       ~base1() {
                 cout << "Destructing base1\n";
      }
 };
class base2 {
      public: base2() {
                cout << "Constructing base2\n";
      }
      ~base2() {
                cout << "Destructing base2\n";
      }
};
class derived: public base1, public base2 {
      public: derived() {
                cout << "Constructing derived\n";
      }
       ~derived() {
                cout << "Destructing derived\n";
       }
};
int main() {
      derived ob;
      return 0;
}
```

```
/*Output*/
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

Overriding Member Function

- When the base class and derived class have member functions with exactly the same name, same return-type, and same arguments list, then it is said to be function overriding
- The function overriding also means when the derived class defines the same function as defined in its base class.
- Through function overriding you can perform runtime polymorphism.
- The function overriding allows you to have the same function in child class which is already defined in the parent class.

Overriding Member Function

```
class  A{
      public:
                void func_1(int a, int b){

        cout<<"Base class\n";
         }
};
class B: public A{
      public:
                void func_1(int a, int b){
                                cout<<"Derived class\n";

      }
};
int main(){
      A obj1;
      B obj;
      obj1.func_1(3,4);
      obj.func_1(5,6);
```

```
/*Output*/
Base class
Derived class
```

Overriding Member Function

```
class  A{
    public:
          void func_1(int a, int b){
                    cout<<"Base class\n";

}
};
class B: public A{
    public:
          void func_1(int a, int b){
                    cout<<"Derived class\n";

    }
};
int main(){
    B obj;
    obj.func_1(3,4);
    obj.func_1(5,6);
    return 0;
}
```

```
/*Output*/
Derived class
Derived class
```

Private, Public, Protected Inheritance

When a class member is declared?

- As public, it can be accessed by any other part of a program.
- As private, it can be accessed only by members of its class.
  - Further, derived classes do not have access to private base class members.
- As protected, it can be accessed only by members of its class.
  - However, derived classes also have access to protected base class members.
  - Thus, protected allows a member to be inherited, but to remain private within a class hierarchy.

Private, Public, Protected Inheritance

When a base class is inherited by use of

- **public, (default)**
  - its public members become public members of the derived class.
  - its protected members become protected members of the derived class.
- **protected,** its public and protected members become protected members of the derived class.
- **private,** its public and protected members become private members of the derived class.

Public and Private Inheritance

```
#include <iostream>
using namespace std;
class A //base class
{
        private:
                int
privdataA;
        protected:
                int
protdataA;
        public:
                int
pubdataA;
};
```

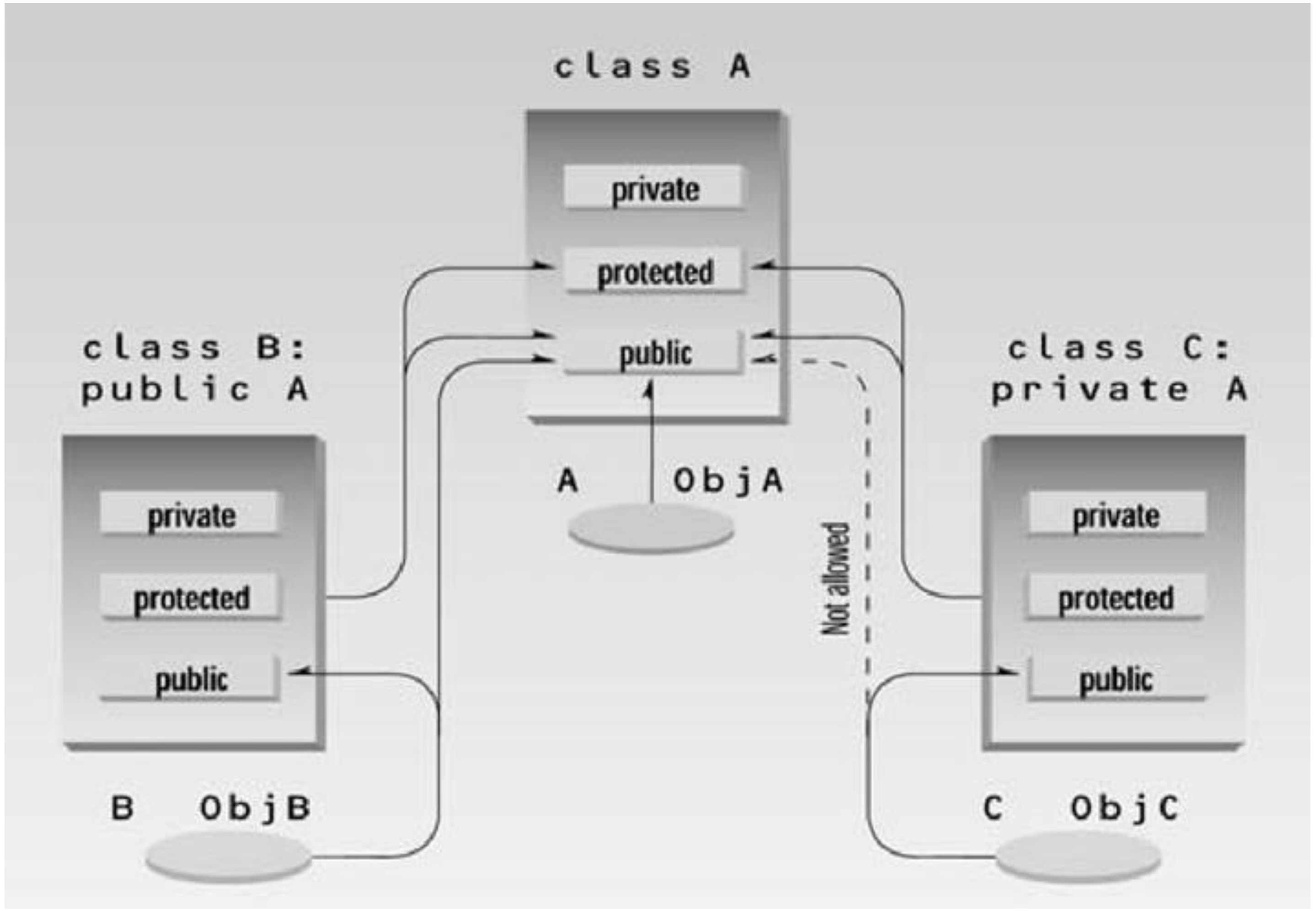
```
class B : public A
{
        public:
                void funct()
                {
                        int a;
                        a =
privdataA; //error
                        a =
protdataA; //OK
                        a =
pubdataA; //OK
                }
};
```

Public and Private Inheritance(Cont..)

```
class C : private A
{
public:
void funct()
{
        int a;
        a = privdataA; //error
        a = protdataA; //OK
        a = pubdataA; //OK
}
};
```

```
int main()
{
        int a;
        B objB;
        a = objB.privdataA;
//error
        a = objB.protdataA;
//error                a =
objB.pubdataA; //OK
        C objC;
        a = objC.privdataA;
//error
        a = objC.protdataA;
//error
        a = objC.pubdataA;
//error
        return 0;
```

Private and Public Inheritance

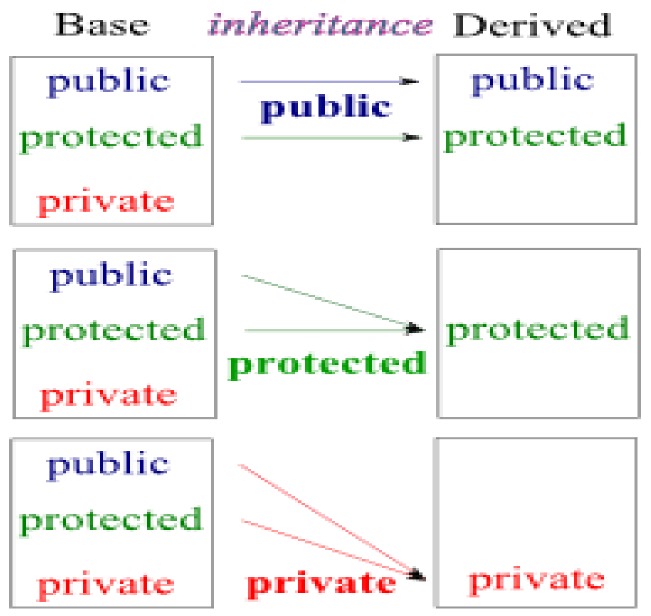


Visibility across access and Inheritance

Inheritance

Visibility

	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private



Inheritance

Reusability

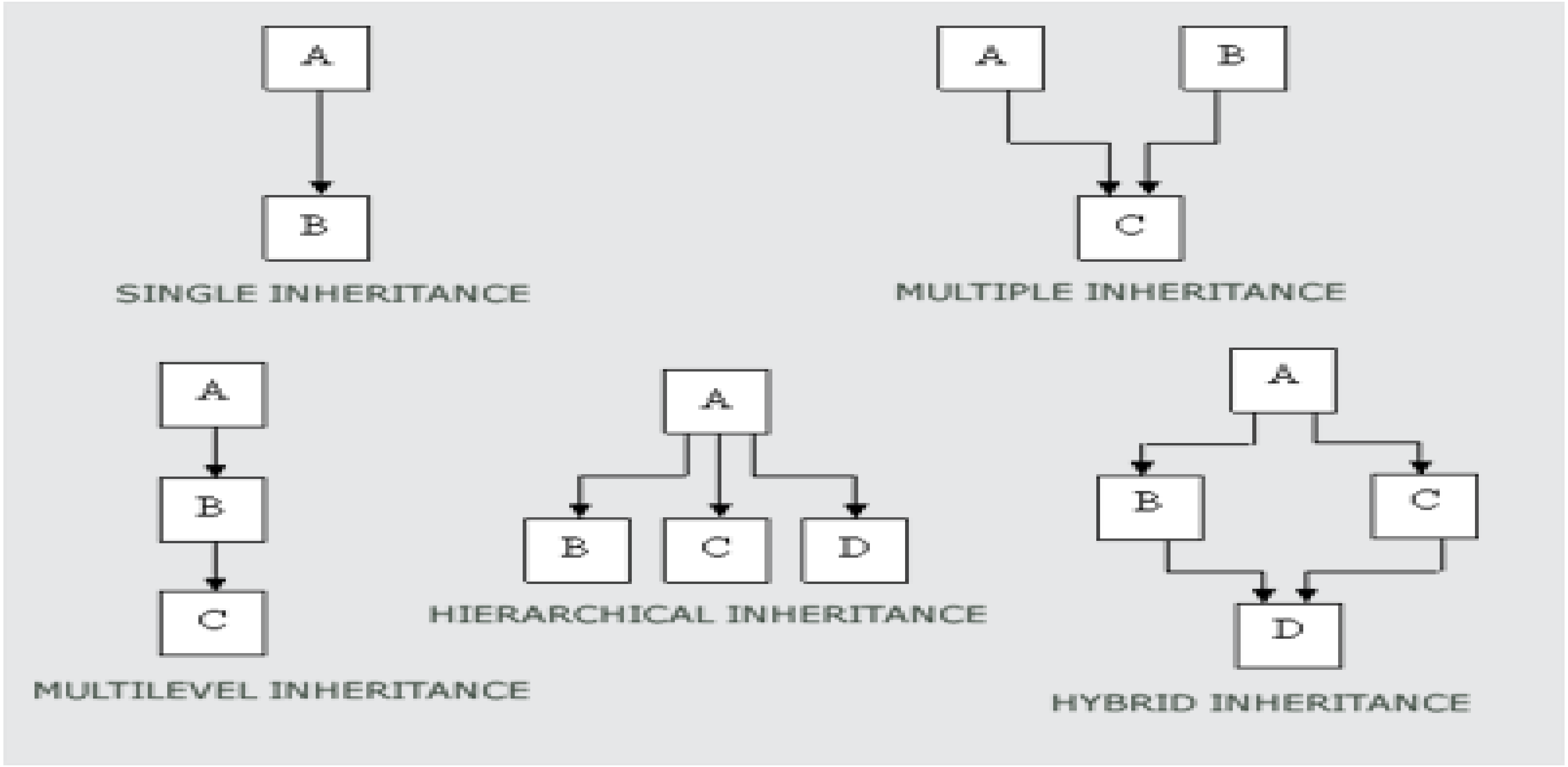
creating new classes and reusing the properties of existing one.

Deriving a new class from the old one.

Types of Inheritance

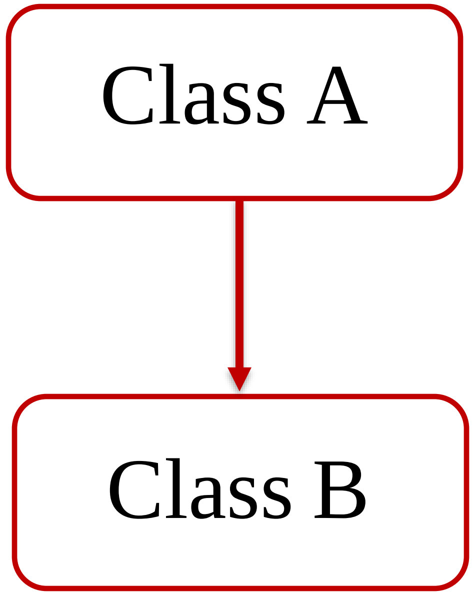
1. Single inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

Types of Inheritance



Single Inheritance

- In single inheritance there exist a single base class and single derived class
- Example: A class Car is derived from the base class Vehicle

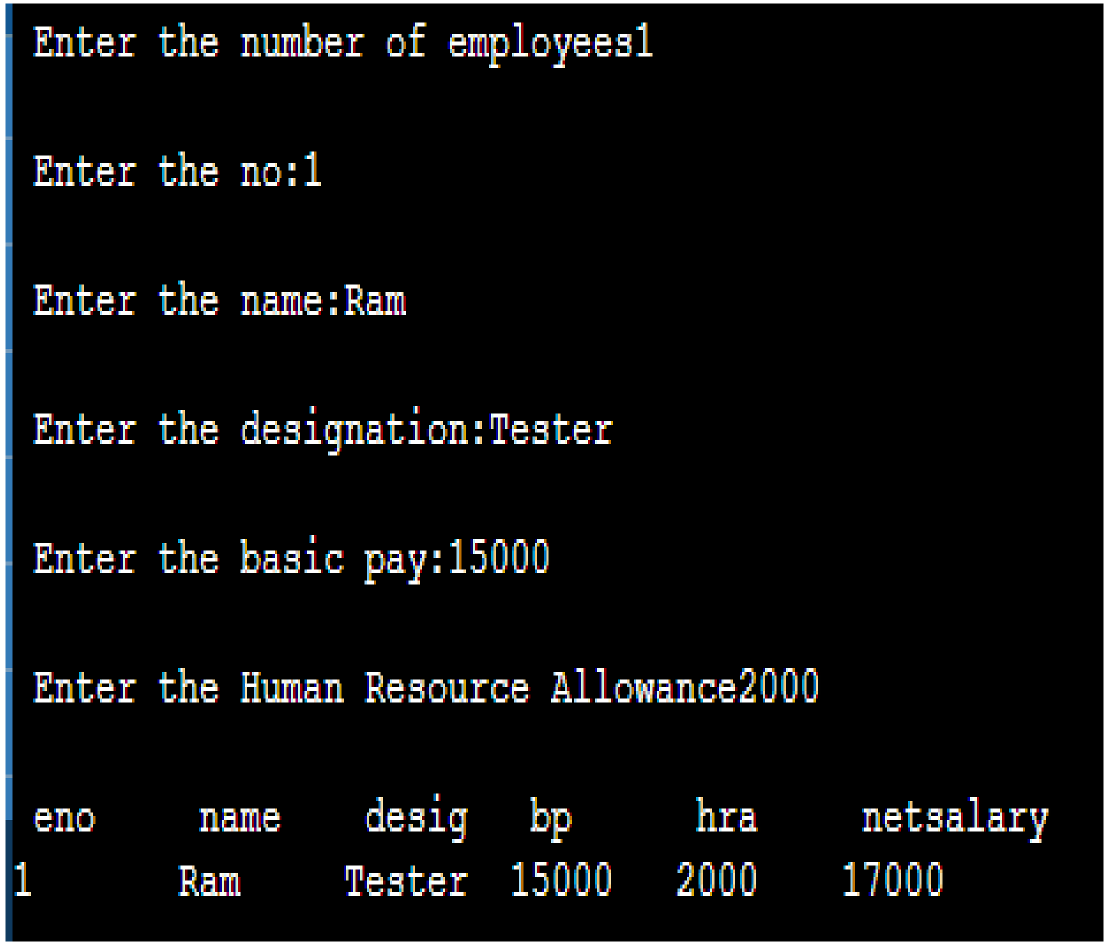


Example of Single Inheritance

```
class emp{
          public: int eno;
                      char name[20], desig[20];
          void getdata(){
                    cout<<"\n Enter the no:";
                    cin>>eno;
                    cout<<"\n Enter the name:";
                    cin>>name;
                    cout<<"\n Enter the
designation:";
                    cin>>des;
          }
} ;
class salary: public emp{
          float basic, hra, netsalary;
          public:
                    void setdata(){
                              cout<<"\n Enter
the basic pay:";
```

```
          cin>>basic;
          cout<<"\n Enter the Human
Resource Allowance";
          cin>>hra;
}
void calculate()
{
          netsalary=basic+hra;
}
void display()
{
          cout<<eno<<"\t"<<name<<"\t
"<<desig<<"\t"<<basic<<"\t"<<hra<<"\
t"<<netsalary<<"\n";
}
};
```

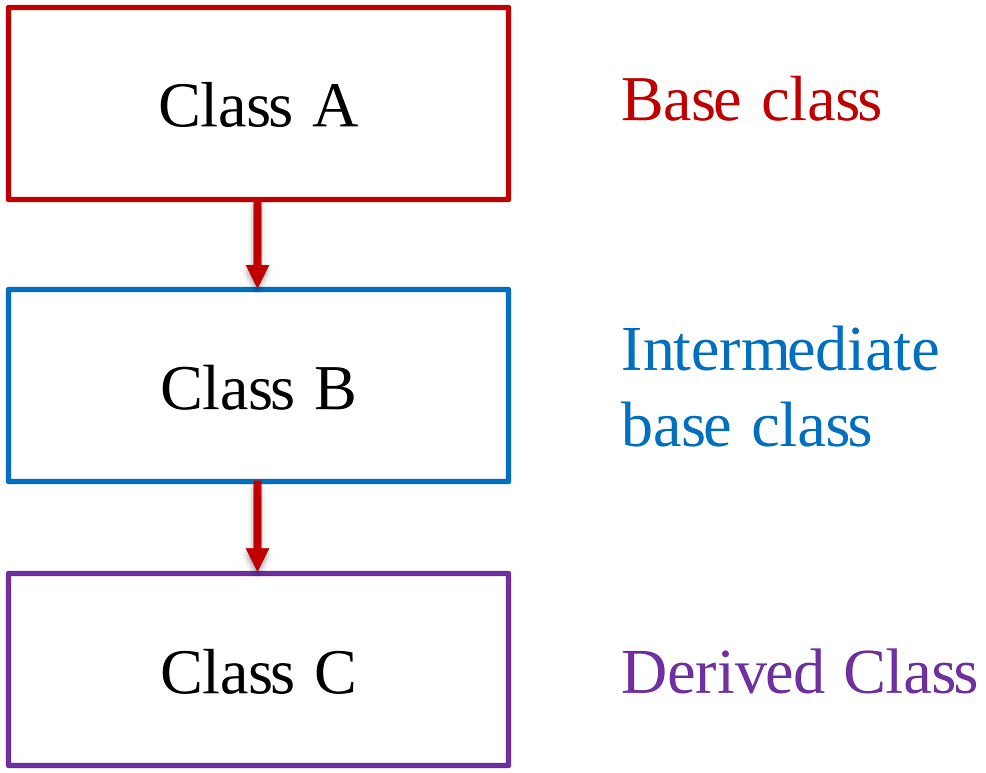
```
int main(){
        int i, no;
        char ch;
        salary s[20];
        cout<<"\n Enter the number of employees";
        cin>>no;
        for(i=0;i<no;i++){
                s[i].getdata();
                s[i].setdata();
                s[i].calculate();
        }
        cout<<"\n eno \t name \t desig \t bp \t hra \t
netsalary\n";
         for(i=0;i<no;i++){
                s[i].display();}
        return 0;
}
```



```
Enter the number of employees1

Enter the no:1

Enter the name:Ram

Enter the designation:Tester

Enter the basic pay:15000

Enter the Human Resource Allowance2000

eno     name    desig   bp      hra     netsalary
1       Ram     Tester  15000   2000    17000
```

Multi level Inheritance

- In multi level inheritance, the derived class can also be derived by and another class, which in turn can further be inherited by another

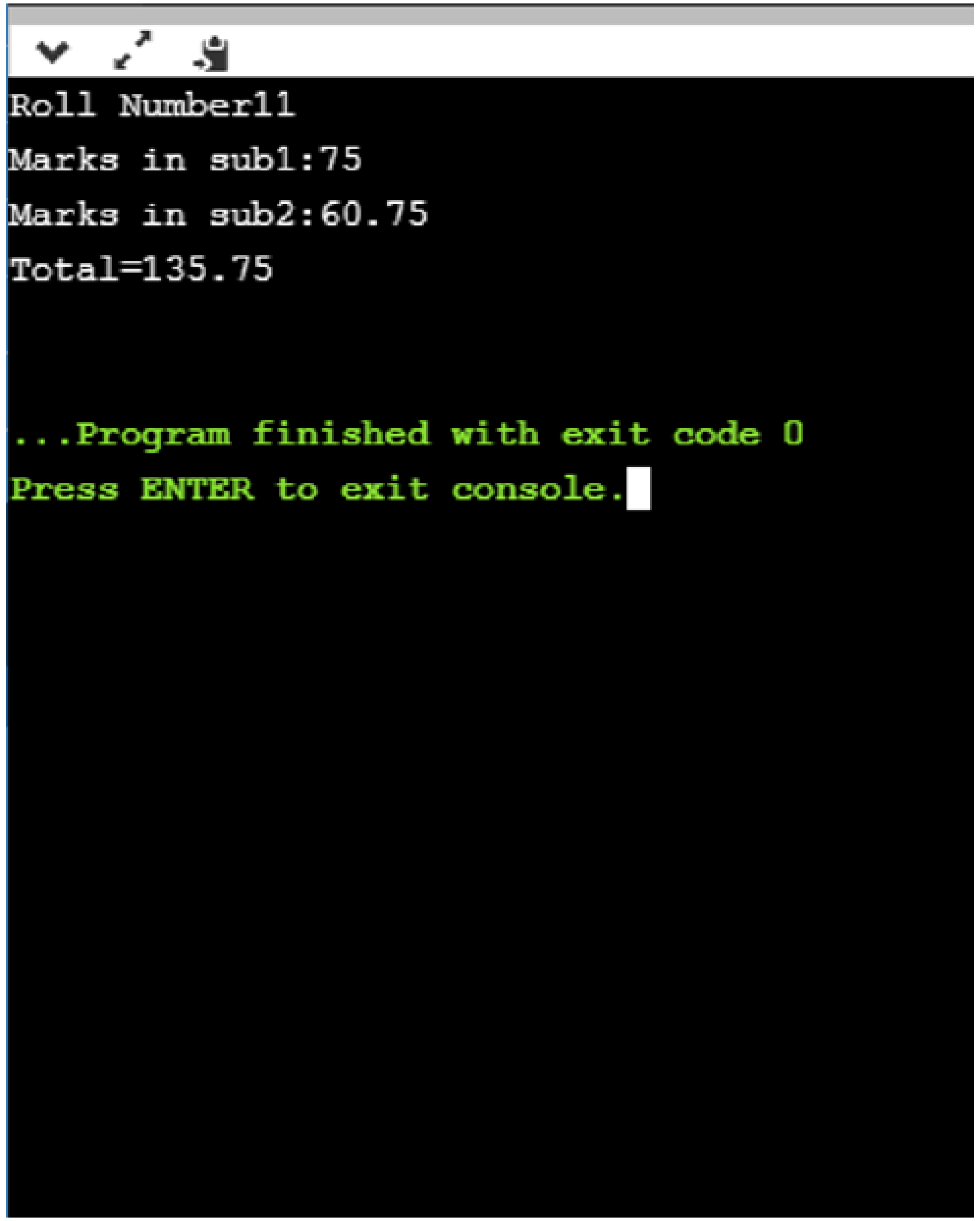


Example:

```
class student{
            protected:
                        int roll_number;
            public:
                        void getdata(int);
                        void putdata();
} ;
void student :: getdata(int a){
            roll_number =a;
}
void student :: putdata(){
            cout<<"RollNumber"<<roll_number<<"\
n";
}
class test: public student{
            protected:
                        float sub1, sub2;
            public:
                        void setdata(float, float);
                        void put_marks();
};
```

```
void test::setdata(float x, float y){
            sub1 = x;
            sub2 = y;
}
void test :: put_marks(){
            cout<<"Marks in
sub1:"<<sub1<<"\n"<<"Marks          in
sub2:"<<sub2<<"\n";
}
class result: public test{
            float total;
            public:
                        void display();
};
void result::display(){
            total = sub1 + sub2;
            putdata();   put_marks();
            cout<<"Total="<<total<<"\n";
}
```

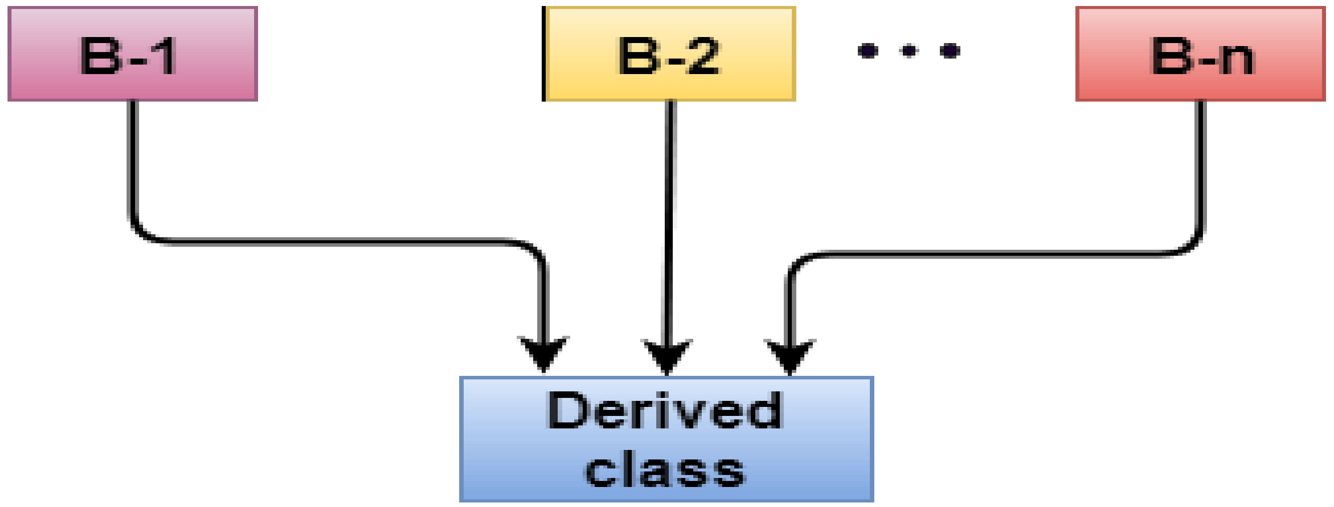
```
int main(){
        result stud;
        stud.getdata(11);
        stud.setdata(75.0,
60.75);
        stud.display();
        return 0;
}
```



```
Roll Number11
Marks in sub1:75
Marks in sub2:60.75
Total=135.75


...Program finished with exit code 0
Press ENTER to exit console.
```

Multiple Inheritance

- In multiple inheritance, a class can inherit more than one class.
- This means that in this type of inheritance a single child class can have multiple parent classes.



Example

```
class M{
    protected: int m;
    public:
    void get_m(int);
};
class N{
    protected: int n;
    public:
    void get_n(int);
};
class P: public M, public N{
    public: void display();
};
```

```
void M::get_m(int  x){
    m= x;
}
void N::get_n(int  y){
    n= y;
}
void P::display(){
    cout<<"m="<<m<<"n="<<n<<"m*n="<<m*n;
}
int main(){
    P p;
    p.get_m(10);
    p.get_n(20);
    p.display();
    return 0;
}
```

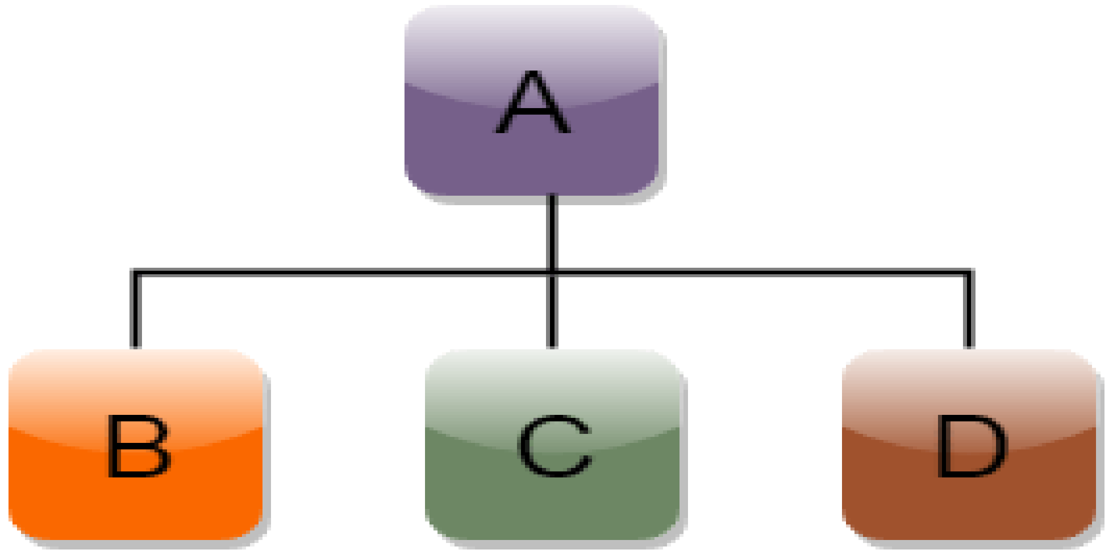
```
/*output*/
m=10n=20m*n=200
```

Ambiguity in Multiple Inheritance

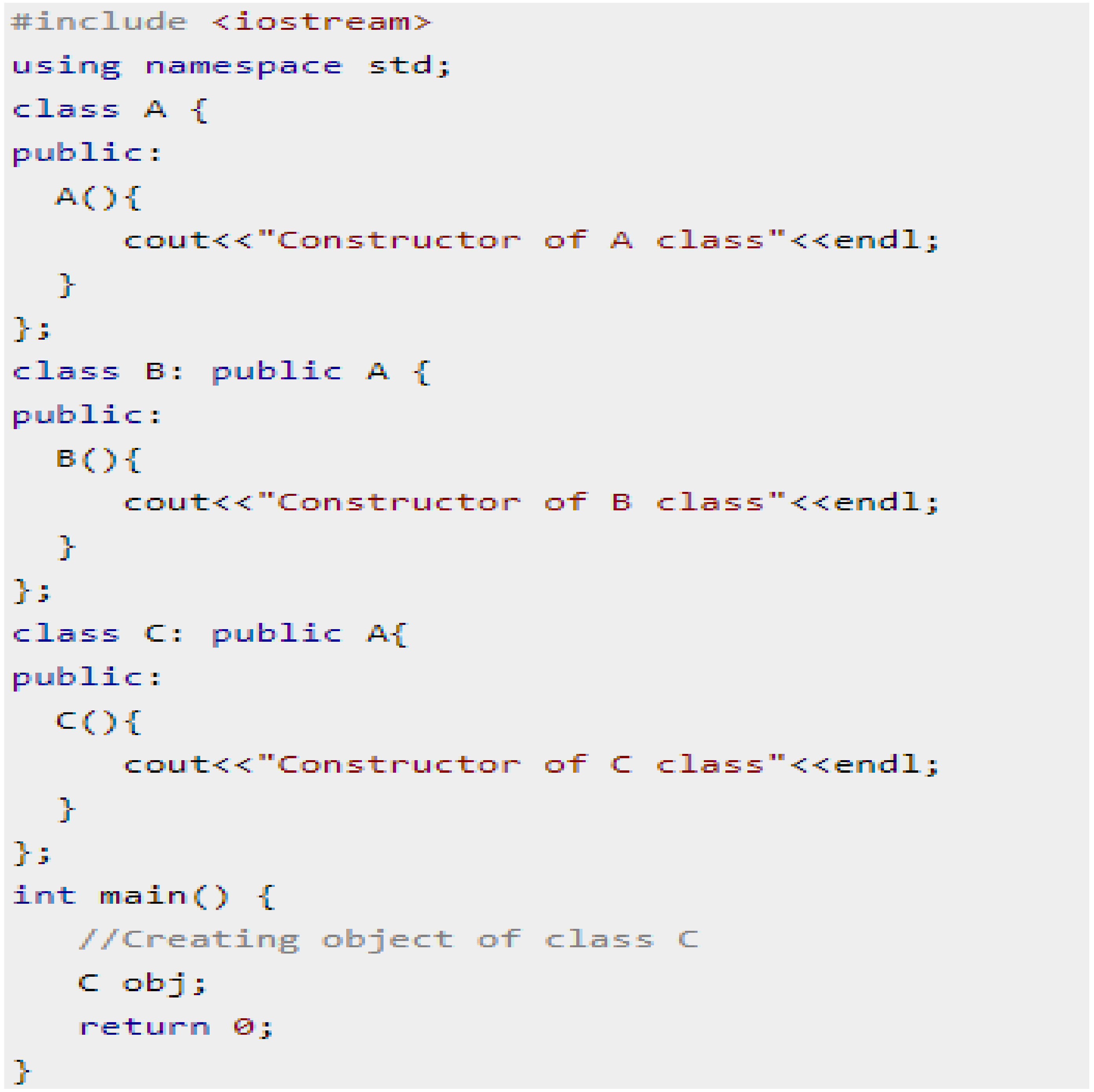
- Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Hierarchical Inheritance

- In this type of inheritance, one parent class has more than one child class.

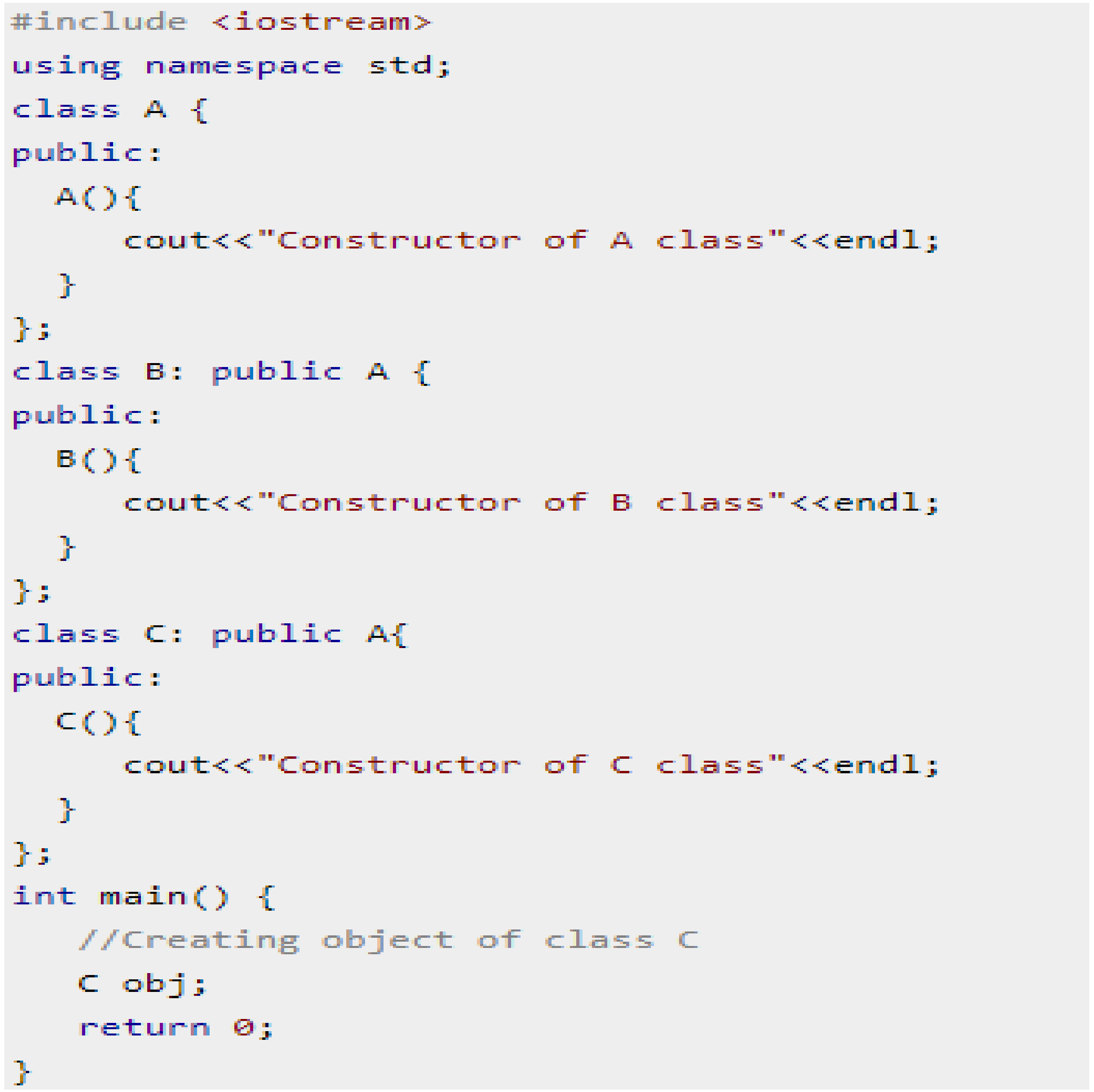


Example



```cpp
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

Example

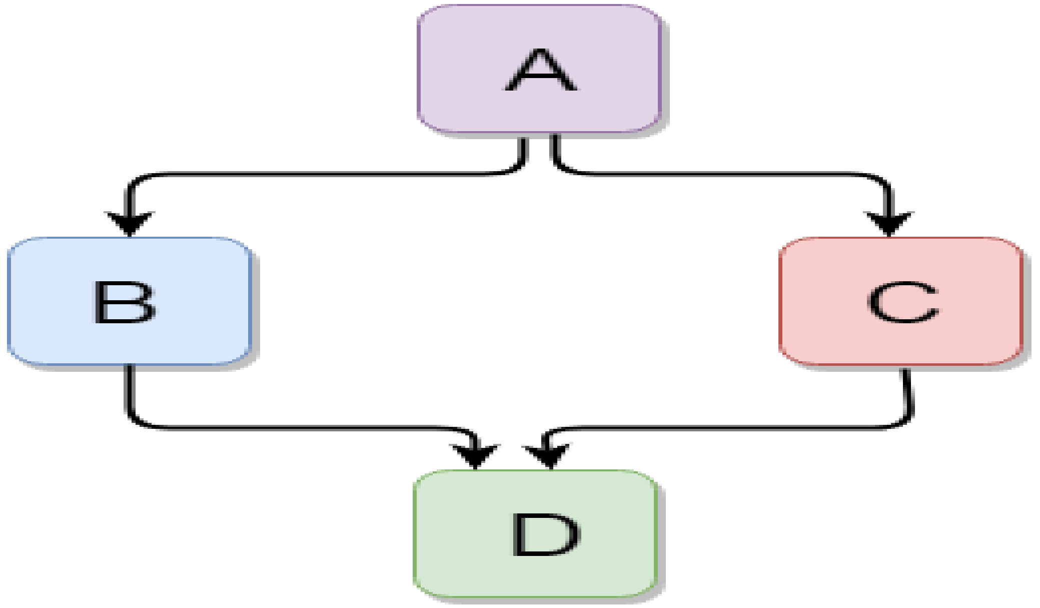


```cpp
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

```
OUTPUT:
Constructor of A class
Constructor of C class
```

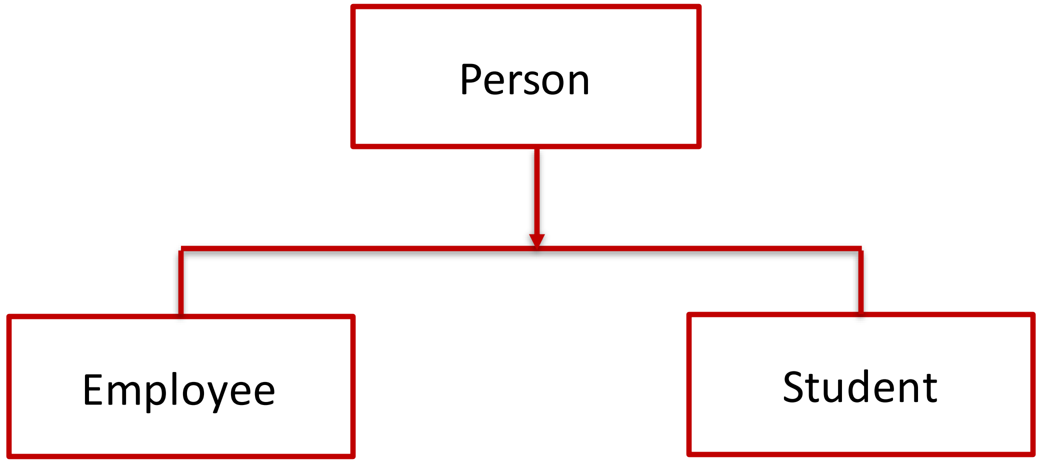
Hybrid Inheritance

- Hybrid inheritance is a combination of more than one type of inheritance.
- For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.



Class Hierarchies

- When more than one classes are derived from a single base classes
- Example:
  - C++ program to create Employee and Student inheriting from Person using Hierarchical Inheritance



Class Hierarchies

```
class person{
   char name[100],gender[10];
   int age;
   public:
   void getdata() {
         cout<<"Name:  ";
        cin>>name;
         cout<<"Age:  ";
         cin>>age;
         cout<<"Gender:  ";
         cin>>gender;
     }
   void display() {
         cout<<"Name:  "<<name<<endl;
         cout<<"Age:  "<<age<<endl;
         cout<<"Gender:  "<<gender<<endl;
     }  };
```

```
class student: public person
{
   char institute[100], year[20];
   public:
      void getdata()
      {
         person::getdata();
         cout<<"Name of College/School:
"<<"\n";
         cin>>institute;
         cout<<"Year:  ";
         cin>>year;
      }
      void display()
      {
         person::display();
         cout<<"Name of College/School:
"<<institute<<endl;
         cout<<"Year:  "<<year<<endl;
      }
};
```

```
void getdata() {
        cout<<"Name: ";
        fflush(stdin);
                cin>>name;
        cout<<"Age: ";
        cin>>age;
        cout<<"Gender: ";
        cin>>gender;
    }
```

```
void display() {
          cout<<"Name:
"<<name<<endl;
          cout<<"Age:
"<<age<<endl;

cout<<"Gender:
"<<gender<<endl;
      }
```

```
class employee: public person {
    char company[100];
    float salary;
    public:
        void getdata() {
            person::getdata();
            cout<<"Name  of Company:  ";
           cin>>company;
            cout<<"Salary:  Rs.";
            cin>>salary;
        }
        void display() {
            person::display();
            cout<<"Name  of Company:  “
                       <<company<<endl;
            cout<<"Salary:
Rs."<<salary<<endl;
        }
};
```

```
void getdata()  {
          cout<<"Name:  ";
          fflush(stdin);
                       cin>>name;
          cout<<"Age:  ";
          cin>>age;
          cout<<"Gender:  ";
          cin>>gender;
     }
```

```
void display()  {
          cout<<"Name:
"<<name<<endl;
          cout<<"Age:
"<<age<<endl;
          cout<<"Gender:
"<<gender<<endl;
      }
```

```
int main()
{
                student  s;
                 employee  e;
                cout<<"Student"<<endl;
                cout<<"Enter  data"<<endl;
                s.getdata();
                 cout<<endl<<"Displaying
data"<<endl;
                s.display();
                cout<<endl<<"Employee"<<endl;
                cout<<"Enter  data"<<endl;
                e.getdata();
                cout<<endl<<"Displaying
data"<<endl;
                 e.display();
                getch();
                return  0;
}
```

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class’s constructor is executed first, followed by the derived class’s constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Constructors and Destructors in Base and Derived Classes

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

//*********************************
// BaseClass declaration          *
//*********************************

```

Constructors and Destructors in Base and Derived Classes

```cpp
class BaseClass
{
public:
   BaseClass()  // Constructor
      { cout << "This is the BaseClass constructor.\n"; }

   ~BaseClass() // Destructor
      { cout << "This is the BaseClass destructor.\n"; }
};

//*********************************
// DerivedClass declaration       *
//*********************************

class DerivedClass : public BaseClass
{
public:
   DerivedClass()  // Constructor
      { cout << "This is the DerivedClass constructor.\n"; }

   ~DerivedClass()  // Destructor
      { cout << "This is the DerivedClass destructor.\n"; }
};

```

Constructors and Destructors in Base and Derived Classes

```
//********************************
// main function                 *
//********************************

int main()
{
   cout << "We will now define a DerivedClass object.\n";

   DerivedClass object;

   cout << "The program is now going to end.\n";
   return 0;
}
```

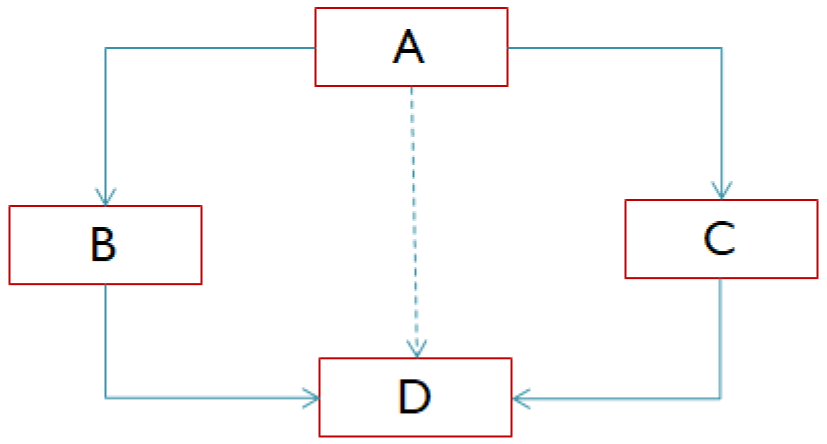
Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

Initializing base class in derived class constructor

```
#include <iostream>
using namespace std;
class Base
{
   int x;
   public:
   // parameterized constructor
   Base(int i)
   {
      x = i;
      cout << "Base Parameterized Constructor\n";
   }
};
class Derived: public Base
{
   int y;
   public:
   // parameterized constructor
   Derived(int j):Base(j)
   {
      y = j;
      cout << "Derived Parameterized Constructor\n";
   }
};

int main()
{
   Derived d(10) ;
}
```

Virtual Base Classes



- In above inheritance, B & C classes inherit features of A & then as D is derived from B & C
- There is **an ambiguity error** because **the compiler does not understand how to pass features of class A to class D either using B class or C class**

Virtual Base Classes

- So to remove the ambiguity you need to make the class B and class C as virtual base classes

```
class B: public virtual A{
 };
class C: public virtual A{
               };
```

- When the class is made as virtual base class, the compiler takes **necessary care to pass only one copy of the members in inherited class** regardless of many inheritance paths exist between the virtual base class and derived class

Virtual Base Classes

```
class student {
  protected: int roll_number;
  public:
  void get_number(int  a) { roll_number  = a; }
  void put_number() {
       cout <<"Roll Number:  "<<roll_number  << "n"; }
 };
class test : public virtual student {
  protected: float part1, part2;
  public:
  void get_marks(float x, float y)
  {   part1 = x; part2 = y; }
  void put_marks()
  {
  cout<<"Markts Obtained: "<<"n"<<"Part1 = "<<part1<<
"n"<<"Part2 = "<<part2<< "n";
  }
};
```

```
class sports : virtual public student {
  protected:  float score;
  public: void get_score(float  s) { score = s; }
  void put_score()  { cout << "Sports Marks  = " << score << "n"; }
};
class result : public test, public sports {
  float total;
  public: void display();
};
void result :: display() {
  total = part1+part2+score;
  put_number();
  put_marks();
  put_score();
  cout<<"Total  Score: "<<total  <<"n"; }
int main() {
  result student1;
  student1.get_number(10);
   student1.get_marks(75.25,82.90);
   student1.get_score(92.80);
  student1.display();
  return 0;
}
```

Abstract Class

- An abstract class is one that is not used to create objects
- Is designed only to act as a base class
- It is a design concept in program development and provide a base upon which other classes can be built

Nesting of Classes

```
class Marks{
          private:
                    int rno;
                    float perc;
                    public:                    //constructor
                              Marks() {
                                        rno = 0; perc = 0.0;
                              }
                    //input roll numbers and percentage
                    void readMarks(void){
                              cout<<"Enter roll number: ";
                              cin>>rno;
                              cout<<"Enter percentage: ";
                              cin>>perc;
                    }
                    //print roll number and percentage
                    void printMarks(void) {
                              cout<<"Roll No.: "<<rno<<endl;

              cout<<"Percentage: "<<perc<<"%"<<endl;
                    }
};
```

```
class Student{
        private:
                //object to Marks class
                Marks objM;
                char name[30];
                public:
                        //input student details
                        void readStudent(void){
              cout<<"Enter name: ";
                         cin.getline(name, 30);

                 objM.readMarks();

            }
                        void printStudent(void){

              cout<<"Name: "<<name<<endl;
              objM.printMarks();
                        }
};
int main(){
        Student std;
        std.readStudent();
        std.printStudent();
        return 0;
}
```

```
Enter name: abc
Enter roll number: 5
Enter percentage:
60
Name: abc
Roll No.: 5
Percentage: 60%
```

Friend Classes in C++

- Similarly like, friend function. A class can be made a friend of another class using keyword friend. For example,

```
 class A
{
   friend class B;

};

class B
{

};
```

Example : Add members of two different classes using friend functions

```
class A {
    private:
        int numA;
        friend class B;
    public:
        A() : numA(12) {}
};
class B {
    private:
        int numB;
    public:
        B() : numB(1) {}
    int add() {
        A objectA;
        return objectA.numA + numB;
    }
};
int main() {
    B objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Thank You