Chapter 5: Inheritance

BIBHA STHAPIT
ASST. PROFESSOR
IOE, PULCHOWK CAMPUS

Introduction

- Re-accessability / Reusability is yet another feature of OOP.
- The C++ classes can be used again in several ways by defining the new classes, reusing the properties of existing ones.
- The mechanism of deriving a new class from an old one is called ‘Inheritance'.
- The inheritance relationships are very common in the real-world scenario. For example, mammal is a class derived from living things, people is a class derived from mammal, man and woman are classes derived from people.

Introduction

- This is often referred to as IS-A' relationship because very object of the class being defined "is" also an object of inherited class.
- The old class is called 'BASE' class and the new one is called 'DERIVED‘ class.
- The new class contains all (or some of)the attributes of the old class in addition to some of its own attributes.
- Private data members cannot be inherited. So, access specifier “protected” is used so that they can be inherited in derived class.

Defining Derived Classes

- A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class derived_classname : Access_specifier base_classname
{
__
__ // members of derived class
};
```

- The colon indicates that the derived-classname is derived from the base_classname.

Defining Derived Classes

- The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private.
- Visibility mode describes the status of derived features e.g.

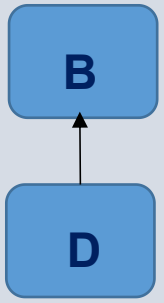
```
class xyz //base class
{
members of xyz
};
class ABC : public xyz //public derivation
{
members of ABC
};
class ABC : XYZ //private derivation (by default)
{
members of ABC
};
```

Defining Derived Classes – Visibility mode

- **1. Private:** when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class. The result is that the member of base class will not be accessible to the objects of the derived class. Therefore the public members of the base class can be accessed by its own objects using the dot operator.
- 2. **Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.
- 3. **Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class

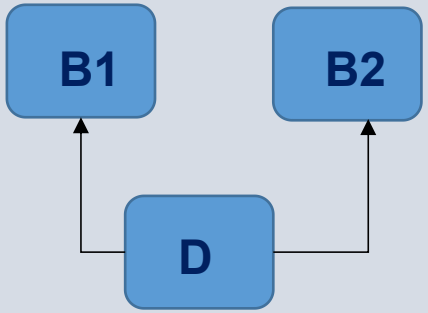
Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

Types of inheritance



Single inheritance:

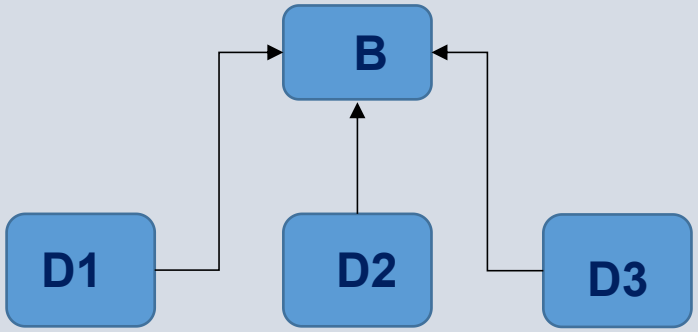
```
class B
 { ……. };
class D : public B
{ …… };

main( )
{ D d; }
```



Multiple inheritance:

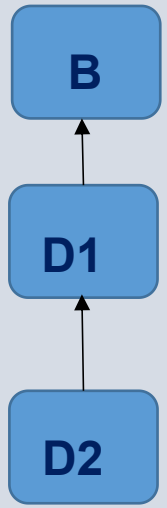
```
class B1
 { ……. };
class B2
 { ……. };
class D : public B1, public B2
{ …… };

main( )
{ D d; }
```

Types of inheritance



Hierarchial inheritance:

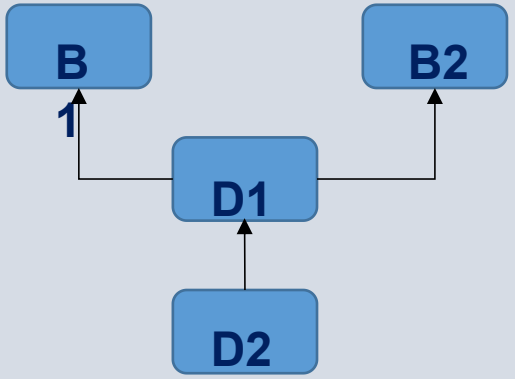
```
class B
 { ……. };
class D1 : public B
 { ……. };
class D2 : public B
{ …… };
class D2 : public B
{ …… };

main( )
{ D1 a;    D2 b;      D3 c;   }
```



Multilevel inheritance:

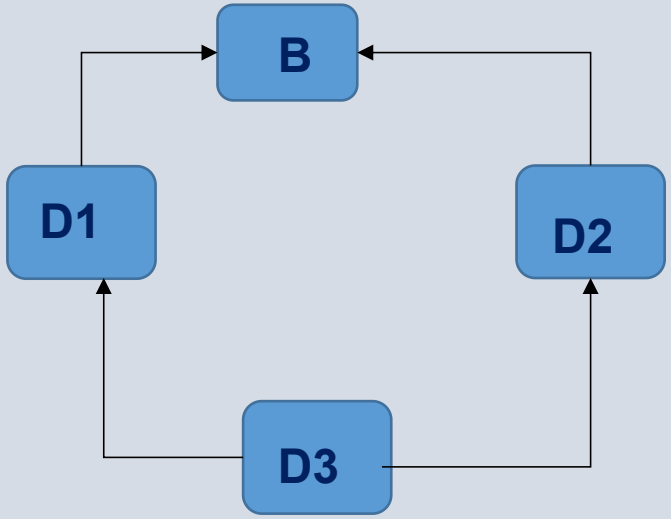
```
class B
 { ……. };
class D1: public B
 { ……. };
class D2: public D1
{ …… };

main( )
{  D2 d;   }
```

Types of inheritance



Hybrid inheritance:

```
class B1
 { ……. };
class B2
 { ……. };
class D1: public B1, public B2
{ …… };
class D2: public D1
{ …… };

main( )
{ D2 d; }
```



Multipath inheritance:

```
class B
 { ……. };
class D1: public B
 { ……. };
class D2: public B
{ …… };
class D3: public D1, public D2
{ …… };

main( )
{ D3 d; }
```

Function Overriding

- Defining a member function in the derived class in such a manner that its name and signature match those of a base class function is known as function overriding.

- Function overriding results in two functions of the same name and same signature. One of them is in the base class. The other one is in the derived class.

- Whenever a function is called with respect to an object of a class, the compiler first searches for the function prototype in the same class. Only if this search fails, the compiler goes up the class hierarchy to look for the function prototype..

Function Overriding

- Of course, the overridden function of the base class will be called if it is called with respect to an object of the base class

- The overridden base class function can still be called with respect to an object of the derived class by using the scope resolution operator

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

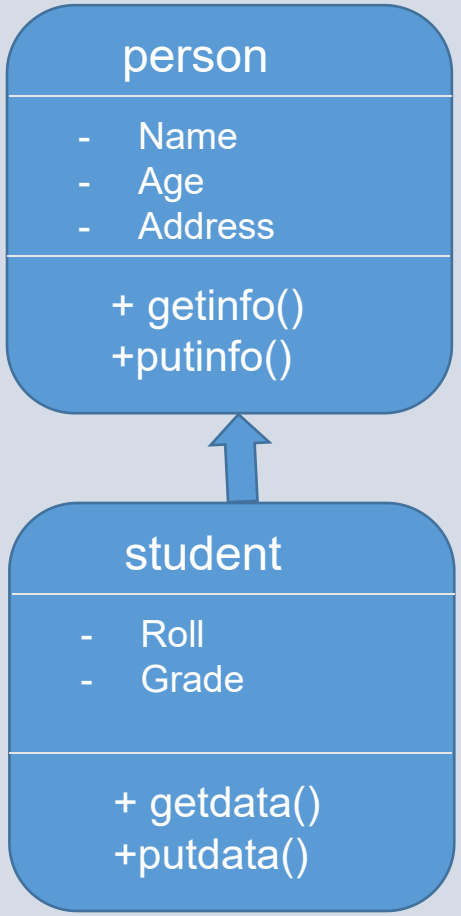
```
class Derived : public Base
{
public:
void show();
}
```

```
main()
{ Derived d;
D.show();
D.base::show();
}
```

Single Inheritance

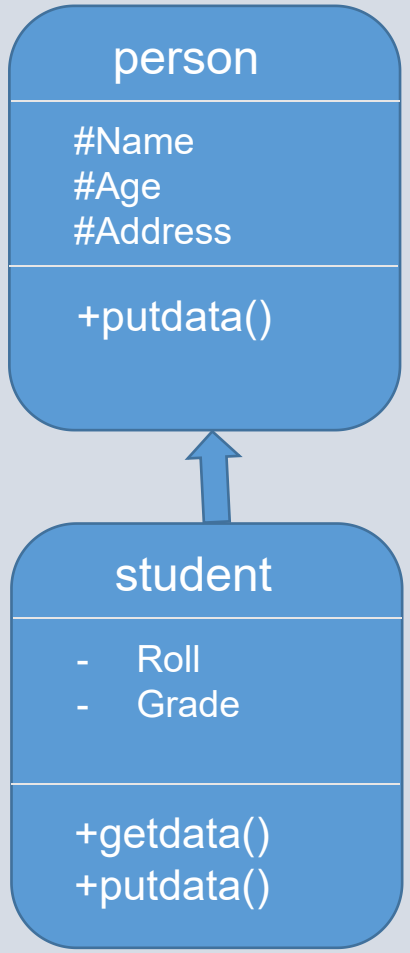
```
class person
{    string name;
    int age;
    string address;
public:
    void getinfo()
    {   cout<<"Enter name:";    getline(cin,name);
        cout<<"Enter age:";    cin>>age;
        fflush(stdin);
        cout<<"Enter address:";  getline(cin,address);
    }
    void putinfo()
    {   cout<<"Name:"<<name<<endl;
        cout<<"Age:"<<age<<endl;
        cout<<"Address:"<<address<<endl;
    }
};
```

```
class student:public person
{    int roll, grade;
public:
    void getdata()
    {   cout<<"Enter roll:";    cin>>roll;
        cout<<"Enter grade:";    cin>>grade;
    }
    void putdata()
    {    putinfo();
        cout<<"Roll:"<<roll<<endl;
        cout<<"Grade:"<<grade<<endl;
    }
};
main()
{    student s;
    s.getinfo();
    s.getdata();
    s.putdata();    }
```



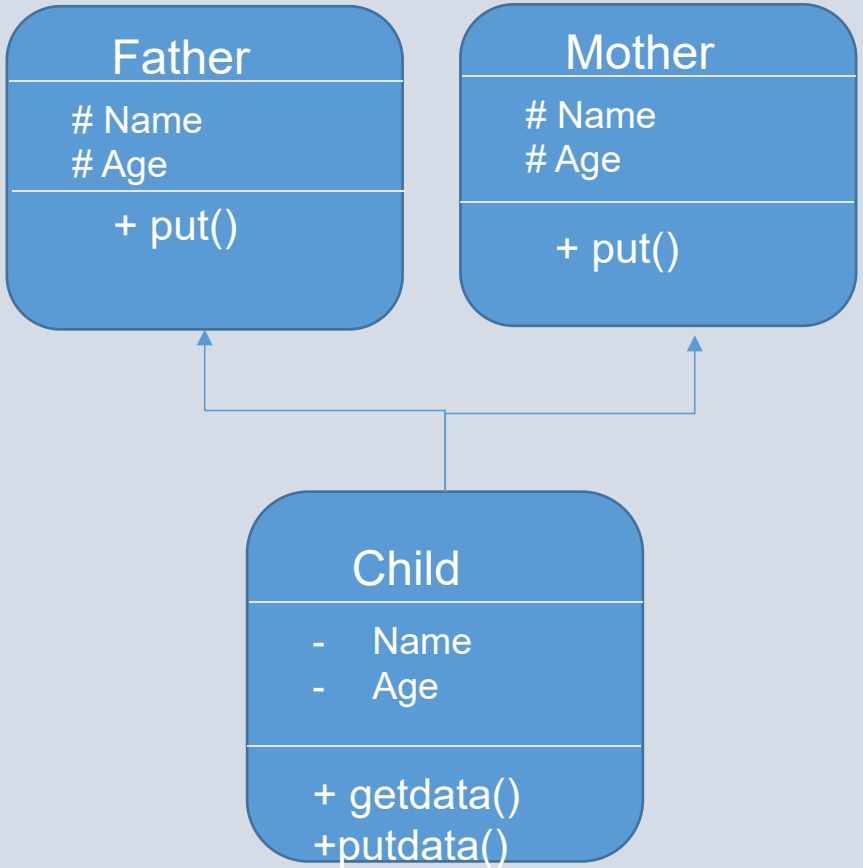
```
class person
{
protected:
   string name;
   int age;
   string address;
public:
   void putdata()
   {
      cout<<"Name:"<<name<<endl;
      cout<<"Age:"<<age<<endl;
      cout<<"Address:"<<address<<endl;
   }
};
```

```
class student:public person
{
   int roll, grade;
public:
   void getdata()
   {
      cout<<"Enter name:";    getline(cin,name);
      cout<<"Enter age:";    cin>>age;
      fflush(stdin);
      cout<<"Enter address:";  getline(cin,address);
      cout<<"Enter roll:";    cin>>roll;
      cout<<"Enter grade:";    cin>>grade;
   }
   void putdata()
   {
      //person::putdata();
      cout<<"Roll:"<<roll<<endl;
      cout<<"Grade:"<<grade<<endl;
   }
};

main()
{
   student s;
   s.getdata();
   s.person::putdata();
   s.putdata();
}
```



Ambiguities in multiple inheritance

- A derived class is created with more than one base class.
- If two or more base classes have same member names, even though derived class has not over-ridden those members, they are accessed through member resolution operator as there will be confusion which version of member and of which class. This situation is considered as ambiguity in multiple inheritance.



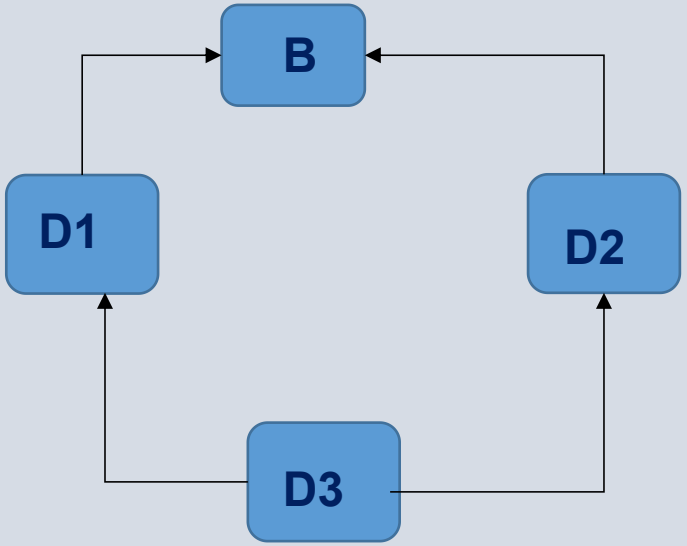
```
class father
{  protected:
    string name;
    int age;
public:
   void put()
    {    cout<<“F.Name:"<<name<<endl;
        cout<<“F.Age:"<<age<<endl;
    }
};

class mother
{  protected:
    string name;
    int age;
public:
   void put()
    {   cout<<“M.Name:"<<name<<endl;
        cout<<“M.Age:"<<age<<endl;
    }
};
```

```
class child : public father, public mother
{    string name;
    int age;
public:
    void getdata()
    {    cout<<"Enter F. name:";
getline(cin,father::name);
        cout<<"Enter age:";    cin>>father::age;
        fflush(stdin);
        cout<<"Enter M. name:";
getline(cin,mother::name);
        cout<<"Enter age:";    cin>>mother::age;
        fflush(stdin);
        cout<<"Enter C. name:";    getline(cin,name);
        cout<<"Enter age:";    cin>>age;
 }
    void putdata()
    {            cout<<"Name:"<<name<<endl;
        cout<<"Age:"<<age<<endl;
  }
};
```

```
main()
{
    child s;
    s.getdata();
   // s.put(); doesn't work
    s.father::put();
    s.mother::put();
    s.putdata();
}
```

Multipath inheritance and virtual base class

- In multipath inheritance (as shown in fig.), the derived class D3 is derived from two consecutive intermediate base classes D1 and D2 which have common base class B.



- This results the data duplication during inheritance , i.e., class D3 will have duplicate data of class B through two paths D1 and D2.

Multipath inheritance and virtual base class

- Hence to avoid data duplication, the base class B is made “virtual” such that only one copy of base class is inherited no matter how many exists.
- Hence,

```
class B
        {…… };
class D1: virtual public B
        {….. };
class D2 : public virtual B
        {….. };
class D3 : public D1, public D2
        {….. };
```

```
class person
{    string name;
    int age;
    string address;
public:
    void getdata()
    {    cout<<"Enter name:";
getline(cin,name);
        cout<<"Enter age:";    cin>>age;
        fflush(stdin);
        cout<<"Enter address:";
getline(cin,address);
    }
    void putdata()
    {    cout<<"Name:"<<name<<endl;
        cout<<"Age:"<<age<<endl;
        cout<<"Address:"<<address<<endl;
    }
};
```

```
class student : public virtual person
{
    int roll;
    int grade;
public:
    void getdata()
    {
        cout<<"Enter roll:";    cin>>roll;
        cout<<"Enter grade:";    cin>>grade;
    }
    void putdata()
    {
        cout<<"Roll:"<<roll<<endl;
        cout<<"Grade:"<<grade<<endl;
    }
};
```

```
class teacher : virtual public person
{    string faculty;
public:
    void getdata()
    {
        fflush(stdin);
        cout<<"Enter faculty";
getline(cin,faculty);
    }
    void putdata()
    {
        cout<<"Faculty:"<<faculty<<endl;
    }
};

class TA : public student, public teacher
{
};
```

```
main()
{
    TA s;
    s.person::getdata();
    s.student::getdata();
    s.teacher::getdata();
    s.person::putdata();
    s.student::putdata();
    s.teacher::putdata();
}
```

Constructors and destructors in inheritance

- If there is only default constructor in base class, then, it is not mandatory to define constructor for the derived class
- If the base class contains parameterized constructor, then, it is mandatory to have constructor in derived class as well.
- Since we usually create object of derived class rather than base class, the derived class provides the argument for the base class.

Constructors and destructors in inheritance

– derived-constructor(arg1, arg2, arg3….. ,argN) : base1(arg1, arg2),

base2(arg3)… , baseN(argN)

{ }

– D(int a1, int a2, float b1, float b2, int c) : A(a1, a2) , B(b1, b2), d1(c) { }
  - Where, A and B are base classes of derived class D
  - a1, a2 -> data members of class A
  - b1, b2 -> data members of class B
  - d1 -> data members of class D

– The object will be created as, D objD(10, 15, 1.35, 4.67, 33)

Constructors and destructors in inheritance

Method of inheritance	Order of execution
Single : class A { … }; class B : public A { … };	A ( ) -> B( )
Multiple : class A { … }; class B { … }; class C : public A, public B { … };	A ( ) -> B( ) -> C ( )
Multiple with virtual: class A { … }; class B { … }; class C : public A, virtual public B { … };	B ( ) -> A ( ) -> C ( )
Multilevel: class A { … }; class B : public A { … }; class C : public B { … }; class D : public C { … };	A ( ) -> B( ) -> C ( ) -> D ( )

```
class base1
{
   int x;
public:
   base1(int i)
   {
      cout<<"Base1 constructor"<<endl;
      x=i;
   }
   void showx()
   {
      cout<<"x="<<x<<endl;
   }
   ~base1()
   {
      cout<<"Base1 destructor"<<endl;
   }
};
```

```
class base2
{   int y;
public:
   base2(int i):y(i)
   {
      cout<<"Base2 constructor"<<endl;
   }
   void showy()
   {
      cout<<"y="<<y<<endl;
   }
   ~base2()
   {
      cout<<"Base2 destructor"<<endl;
   }
};
```

```
class derived : public base1, public virtual base2
{
   int z;
public:
   derived (int a, int b, int c): base1(a), base2(b), z(c)
   {
      cout<<"Derived constructor"<<endl;
   }
   void showz()
   {
      cout<<"z="<<z<<endl;
   }
   ~derived()
   {
      cout<<"Derived destructor"<<endl;
   }
};
```

```
main()
{    derived d(12, 24, 36);
   d.showx();
   d.showy();
   d.showz();
}
```

Output:

```
Base2 constructor
Base1 constructor
Derived constructor
x=12
y=24
z=36
Derived destructor
Base1 destructor
Base2 destructor
```