Unit 5: Inheritance [7hrs]

Reusability:

Reusability is yet another important feature of OOP. It is always nice if we would reuse something that already exists rather than trying to create the same thing all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adopted by other programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. This mechanism of deriving a new class from an old one is called inheritance.

Introduction to Inheritance

- The mechanism of deriving a new class from an old class is called inheritance.
- It provides the use of reusability.
- C++ class es can be re-used using inheritance
- The derived class inherites the some of or all of the properties of the base class .
- A derived class with only one base class is called single inheritance.
- A class can inherit properties from more than one class which is known as multiple inheritance.
- A class can be derived from another derived class which is known as multilevel inheritance.
- When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.

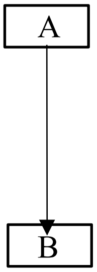


Fig.
Single inheritance

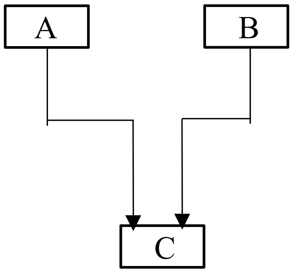


Fig.
Multiple inheritance

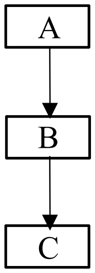


Fig.
Multilevel Inheritance

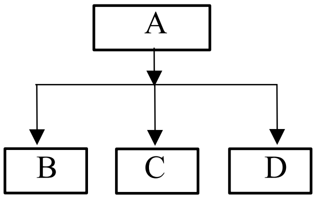


Fig. Hierarchical

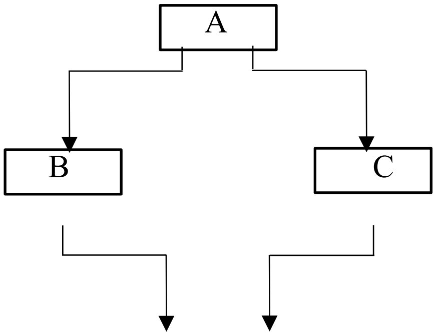


Fig. Forms of Inheritance

Defining Derived class :

Syntax:

```
class  derived – class _name:: visibility-mode base-class
{
……………………..
……………………
……………………
// members of derived class
};
```

Example:

```
class ABC: private xyz // private derivation
{
Member of ABC
};
class ABC: public xyz // public derivation
{
Members of ABC
};
class ABC: xyz // private derivation by default
{
Members of ABC
};
```

Single Inheritance:

a. public derivation

```
#include<iostream>
using namespace std;
class B
{
    int a;                  //private, not inheritable
    public:
    int b;
    void get_ab();
    int get_a (void);
    void show_a(void);
};
class D: public B       // public derivation
{
    int c;
    public:
    void mul(void);
    void display(void);
};
//……………………
void B:: get_ab(void)
    {
    a = 5; b = 10;
    }
int B:: get_a()
    {
    return a;
    }
void B:: show_a()
    {
    cout<<"a="<<a<<"\n";
    }
void D:: mul()
    {
    c = b * get_a(); // a is private can not be inherited
    }
void D:: display()
    {
    cout<<"a="<<get_a()<<"\n";
    cout<<"b="<<b<<"\n";
    cout<<"c="<<c<<"\n";
    }
int main()
{
    D d;
    d.get_ab();
    d.mul();
    d.show_a();
    d.display();
    return 0;
}
```

OUTPUT

```
a = 5
a = 5
b =10
c= 50
```

b. Single inheritance private derivation:

```
# include <iostream>
using namespace std;
class B
{
    int a;
public:
```

```
int b;
 void get_ab();
 int get_a(void);
void show_a(void);
 };

class D: private B
{
      int c;
      public:
      void mul (void);
      void display (void);
};
// ………………………..
void B:: get_ab(void)
      {
      cout<<” Enter value for a and b”;
      cin>> a >>b;
      }
int B:: get_a()
      {
      return a;
      }
void B:: show_a ()
      {
      cout <<”a = “ << a <<”\n”;
      }
void D:: mul ()
      {
      get_ab();
      c =b * get_a(); // a is private
      }
void D:: display()
{ show_a();
cout<<”b=”<<”\n”;
cout<<”c=”<<c<<”\n”;
}
int main()
{ D d;
// d.get_ab(); won’t work
d.mul();
// show_a(); won’t work
d.display();
//d.b = 20; won’t work b is private
```

OUTPUT

```
Enter values for a and b: 5 10
a = 5
b = 10
c = 50
```

```
return 0;
}
```

Making a private member inheritable:

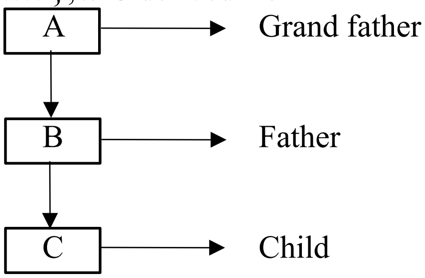
It is seen that a private member of base class cannot be inherited and therefore it is not available for the derived class directly. If the private data needs to be inherited by a derived class , this can be accomplished by modifying the visibility limit of the “private” member by making it “public”. But this would make is accessible to all the other functions of the program, thus taking away the advantage of data hiding. For this, C++ provides a third visibility modifier “protected”, which serve a limited purpose in inheritance. Thus, a member declared as “protected” is accessible by the member functions within its class and any class immediately derived from it.

Visibility of inherited members:

Base class Visibility	**Derived class visibility**		
	Public derivation	**Private derivation**	**Protected derivation**
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Multilevel Inheritance:

```
class A { ……………..}; // base class
class B: public A { ………….. }; // B derived from A
class C: public B { …………… }; // C derived from B
```



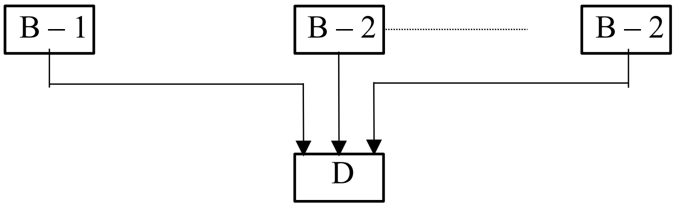
```
# include <iostream>
using namespace std;
class student
{
protected:
int roll_number;
public:
void get_number (int);
void put_number (void);
};

void student:: get_number (int a)
        {
          roll_number = a;
         }
void student:: put_number()
        {
        cout << "roll number:" << roll_number <<"\n";
        }

class test: public student // first derivation
{
protected:
float sub1;
float sub2;
public:
void get_marks (float, float);
void put_marks (void);
};

void test:: get_marks (float x, float y)
           {
           sub1 = x;
           sub2 = y;
          }
void test:: put_marks ()
        {
        cout << "marks in sub1 = "<< sub1 <<"\n";
        cout << "marks in sub2 = "<< sub2 <<"\n";
}
class result: public test           // second derivation
{
float total;
public:
void display (void);
};

void result:: display (void)
        {
         total = sub1 + sub2;
         put_number ();
         put_marks();
        cout << "\n total = " << total;
        }
int main ()
{
result student 1;
```

```
student1.get_number (111);
student1.get_marks (75.0, 59.5);
student1.dsiplay();
}
```

output:

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

Multiple Inheritance:



In multiple inheritance, a class can inherit the attributes of two or more class es.

```
class  D: visibility B1, visibility B-2, ………………
{
………………..
………………..
……………….. // body of D
};
```

Eg.

```
# include <iostream>
using namespace std;
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);
};
void M:: get_m(int x)
        {
        m = x;
        }
void N:: get_n(int y)
        {
        n=y;
        }
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;
}
```

Ambiguity Resolution in inheritance:

If same function name occurs is base and derived class, then ambiguity arises. To avoid this problem, we use scope resolution operator with the function.

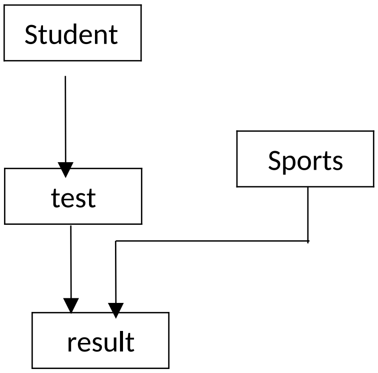
Eg:

```
class M
{
public:
void display (void)
        {
        cout<<” class M\n”;
        }
};
class N
{
public:
          void display (void)
          {
          cout <<"class N\n";
          }
};
class P: public M, public N
{
public:
void display(void) //      overrides display () of M and N
          {
          M:: display ();
          }
};
int main ()
{
          P p;
          p.display();
return 0;
}
```

Output:

```
class  M
```

Hybrid Inheritance:



```
class  sports
{
protected:
float score;
public:
void get_score(float);
void put_score (void)
};
class  result: public test, public sports
{
……………….
……………….
……………….
};
class  test:  public student
{
```

```
……………….
……………….
……………….

};
```

Constructors in derived classes:

- As long as no base class constructors takes any arguments, the derived class need not have a constructor function.
- If any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have constructor and pass the arguments to the base class constructors.
- When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

• In case of multiple inheritance, the base class are constructed "in the order in which they appear in the declaration of the derived class." Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.

Execution of base class constructor

Method 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 B(); ordinary base A(); derived

Example

```
# include <iostream>
using namespace std;
class  alpha
{
int x;
public:
        alpha (int i)
        {
        x = i;
        cout <<"alpha initialized \n";
        }
        void show_x(void)
        {
        cout<<"x = "<< x <<"\n";
        }
};
class  beta
{
float  y;
public:
        beta (float j)
        {
         y = j;
        cout<<"beta initialized\n";
        }
        void show_y (void)
        {
        cout <<"y="<<y <<"\n";
        }
};
class  gamma: public beta, public alpha // order of execution
{
int m,n;
public:
        gamma (int a, float b, int c, int d): alpha (a), beta (b)
        {
         m = c;
        n = d;
        cout<<"gamma initialized\n";
        }
        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();
```

}

Output:

```
beta initialized
alpha initialized
gamma initialized
x = 5
y = 10.75
m = 20
n = 30
```

Note: Beta is initialized first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also, alpha (a) and beta (b) are function calls.

Destructor in derived class

```
class C: public A, public B
 {
   //...
 };
```

1. Here, A class in inherited first, so constructor of class A is called first then the constructor of class B will be called next.

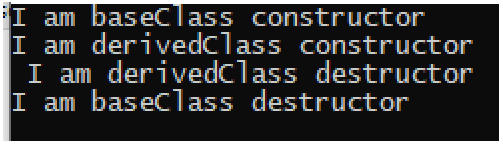
2. The destructor of derived class will be called first then destructor of base class which is mentioned in the derived class declaration is called from last towards first sequence wise.

Example1

```
#include<iostream>
using namespace std;
class baseClass
{
public:
  baseClass()
  {
    cout << "I am baseClass constructor" << endl;
  }

  ~baseClass()
  {
    cout << "I am baseClass destructor" << endl;
  }
};
```

```
class derivedClass: public baseClass
{
public:
  derivedClass()
  {
    cout << "I am derivedClass constructor" << endl;
  }

  ~derivedClass()
  {
    cout <<" I am derivedClass destructor" << endl;
  }
};

int main()
{
  derivedClass D;
  return 0;
}
```

Output



```
I am baseClass constructor
I am derivedClass constructor
 I am derivedClass destructor
I am baseClass destructor
```

Example 2

```
#include <iostream>
using namespace std;
class  alpha
{
int x;
public:
            alpha (int i)
            {
            x = i;
            cout <<"alpha initialized \n";
            }
            void show_x(void)
            {
            cout<<"x = "<< x <<"\n";
            }
~alpha ()
            {

            cout <<"\n------------alpha destroyed--------------";
```

```
        }

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

~beta ()
        {

        cout<<"\n----------beta destroyed--------------";
        }

};
class  gamma: public beta, public alpha // order of execution
{
int m,n;
public:
        gamma (int a, float b, int c, int d): alpha (a), beta (b)
        {
         m = c;
        n = d;
        cout<<"gamma initialized\n";
        }
        void show_mn(void)
        {
        cout<<"m="<<m<<"\n"
        <<"n = " << n <<"\n";
        }
~gamma ()
        {
        cout<<"\n-----------gamma destroyed-------------------";
        }

};
```

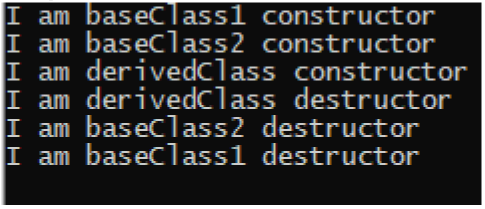
```
int main ()
{
            gamma g(5, 10.75, 20, 30);
            g.show_x();
            g.show_y();
            g.show_mn();
}
```

Destructor in Multiple Inheritance

```
#include<iostream>
using namespace std;
class baseClass1 {
 public:
  baseClass1() {
   cout<<"I am baseClass1 constructor"<<endl;
  }

  ~baseClass1()
  {
   cout<<"I am baseClass1 destructor"<<endl;
  }
 };

 class baseClass2 {
  public:
   baseClass2() {
    cout<<"I am baseClass2 constructor"<<endl;
   }

   ~baseClass2() {
    cout<<"I am baseClass2 destructor"<<endl;
   }
  };

 class derivedClass: public baseClass1, public baseClass2 {
  public:
   derivedClass() {
    cout<<"I am derivedClass constructor"<<endl;
   }

   ~derivedClass() {
    cout<<"I am derivedClass destructor"<<endl;
   }
 };
```

```
int main() {
 derivedClass D;
 return 0;
}
```

Output



```
I am baseClass1 constructor
I am baseClass2 constructor
I am derivedClass constructor
I am derivedClass destructor
I am baseClass2 destructor
I am baseClass1 destructor
```

Member classes: (Nesting of classes):

- Inheritance is the mechanism of deriving certain property of one class into another
- A member can contain object of other classes as its member as shown below:

```
class  alpha { …………… };
class  beta { ……………. };
class  gamma
{
        alpha a;       // a is an object of class alpha
        beta b;        // b is a object of class beta
 }
```

- All objects of gamma class will contain the objects a and b. this kind of relationship is called containership or nesting.
- A class contain object of another class. This is known as containership or nesting.

Example

```
#include<iostream>
#include<conio.h>
const int len=20;
using namespace std;
class student
{
    private:
          char school[len];
          char degree[len];
   public:
        void getdata()
        {
            cout<<"Enter name of the school or university:";
            cin>>school;
            cout<<"Enter highest degree earned:";
            cin>>degree;
        }
        void putdata()
        {
            cout<<endl<<"School or university:"<<school<<endl;
            cout<<endl<<"Highest degree earned:"<<degree<<endl;
        }
};
class employee
{
    private:
            char name[len];
            unsigned long number;
    public:
          void getdata()
          {
              cout<<"Enter name of employee:";
              cin>>name;
              cout<<"Enter number:";
              cin>>number;
          }
          void putdata()
          {
              cout<<endl<<"Name:"<<name;
              cout<<endl<<"Number:"<<number<<endl;
          }
};
class manager
{
    private:
            char title[len];
            double dues;
            employee emp;
            student stu;
    public:
          void getdata()
          {
              stu.getdata();
              cout<<"Enter title:";cin>>title;
              cout<<"Enter golf club dues:";cin>>dues;
              emp.getdata();
          }
          void putdata()
```

```
            {
                stu.putdata();
                cout<<"Title:"<<title;
                cout<<"Gulf club dues:"<<dues;
                emp.putdata();
            }
};
int main()
{
    manager m;
    m.getdata();
    m.putdata();
    getch();
    return 0;
}
```

Aggregation (HAS-A Relationship)

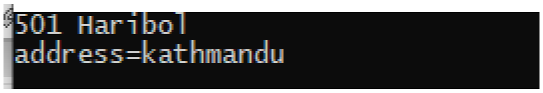
In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>

#include<string.h>

using namespace std;

class Address {

    public:

   char add[30];

     Address(char a[])

    {

        strcpy(add,a);
   }
};
class Employee
    {
        private:
        Address* address;  //Employee HAS-A Address. Example of Aggregation
        public:
        int id;
        char name[20];
        Employee(int i, char j[], Address* add)
       {    id=i;
          strcpy(name,j);
          address=add;
       }
     void display()
       {     cout<<id <<" "<<name<< " "<<"\n";
           cout<<"address="<< address->add;
       }
   };
int main(void) {
    char a[]="kathmandu";
    char b[]="Haribol";
    Address a1= Address(a);
    Employee e1 = Employee(501,b,&a1);
           e1.display();
   return 0;
}
```

<u>Output:</u>



```
501 Haribol
address=kathmandu
```