Unit-3

Inheritance :-

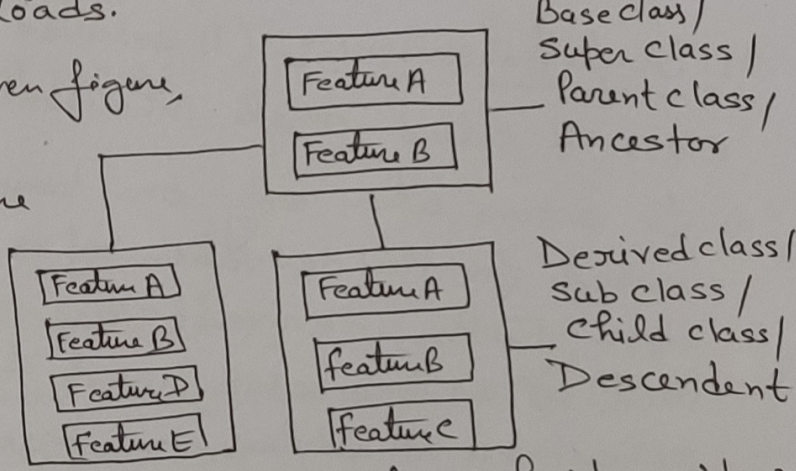
⇒ The idea of classes leads to the idea of inheritance. In our daily lives, we use the concept of classes as divided into subclasses.

⇒ for ex- We know that the class of vehicles is divided into cars, buses, trucks, motorcycles and so on.

⇒ The principle in this sort of division is that each subclass shares common characteristics with the class from which it is derived.

⇒ Cars, trucks, buses and motorcycles all have wheels and a motor; these are the defining characteristics of vehicles. In addition to the characteristics shared with other members of the class, each subclass also has its own particular characteristics: Buses, for instance, have seats for many people, while trucks have space for hauling heavy loads.

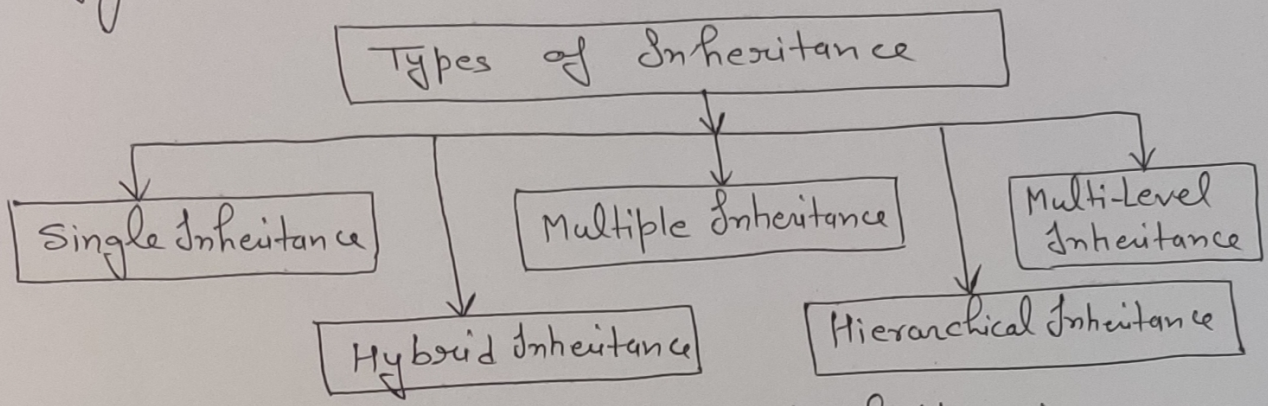
⇒ This is shown in given figure, Notice in figure that feature A and B, which are part of the base class, are common to all the derived classes, but that each derived class also has features of its own.



(fig:- inheritance idea)

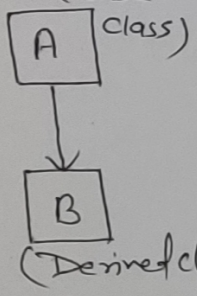
⇒ Inheritance is the process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process.

Types of Inheritance:- The mechanism of deriving a new class from an old one is called inheritance or derivation. The old one is the base class & the new one is called derived class or sub class. there are various forms of inheritance as shown in given figure -



(fig:- Various forms of inheritance)

(a) Single Inheritance:- (A derived class with only one base class is called single inheritance)

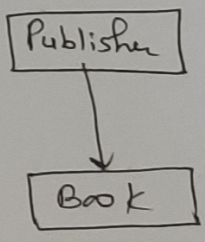


(fig:- single Inheritance)

⇒ It occurs when only one base class is used for the derivation of a derived class. Further, derived class is not used as a base class, such type of Inheritance that has one base and derived class is known as single Inheritance.

⇒ Here, A is a base class and B is derived class shown in the figure.

Example:- let's consider a publication system & assume that the class 'Book' derives name and place of publisher from the Base class "Publisher" then the relationship will be as shown in figure.

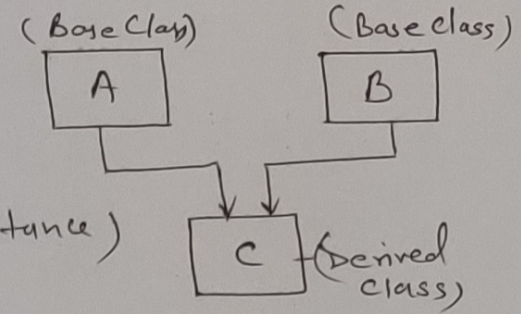


⇒ A Program to implement the concept of single inheritance is shown below -

// Single Inheritance Example

```
#include<iostream>
using namespace std;
class Publisher
{
   string pname;
   string place;
   public:
   void getdata()
    {
     cout<<" Enter name and place of publisher:"<<endl;
     cin>>pname>>place;
    }
    void show()
    {
     cout<<"Publisher Name:"<< pname <<endl;
     cout<<"Place:" << place << endl;
    }
};
Class Book : public Publisher
 {
    string title;  int pages;
     float price;
     public:
     void getdata()
      {
        publisher :: getdata();
        cout<<" Enter Title, Price and No. of Pages"<<endl;
        cin>> title >> price >> pages;
      }
     void show()
      {
        Publisher :: show();
        cout<<"Title:" << title << endl;
        cout<<"Price:" << price <<endl;
        cout<<" No. of Pages:" << pages <<endl;
      }
 };
int main() {  Book b;
              b.getdata(); b.show();
              return 0;
            }
```

(b) Multiple Inheritance:- A class can be derived from more than one base class. This is called multiple inheritance. Given figure shows how this looks when a class C is derived from base classes A and B.



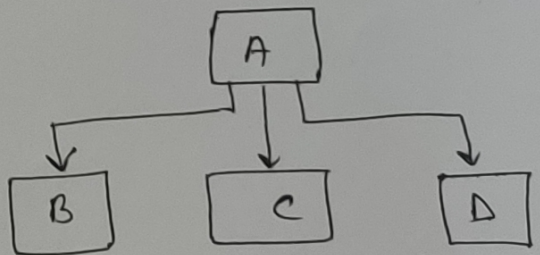
(fig:- Multiple Inheritance)

```
#include <iostream>
using namespace std;
class Publisher
{
 string pname;
 string place;
 public:
  void getdata()
  {
  cout<<"Enter name and place :"<<endl;
  cin>> pname >> place;
  }
  void show()
  {
  cout<<"Publisher Name :" << pname <<endl;
  cout << "Place :" << place <<endl;
  }
};
class Author
 { string aname;
   public:
   void getdata()
   { cout <<"Enter Author Name :"<<endl;
    cin >> aname;
   }
   void show()
    { cout <<"Author Name :" << aname <<endl;
   }
```

```
Class Book : public Publisher, public Author
{
    string title;
    float price;
    int pages;
    public:
    void getdata()
     { Publisher :: getdata();
       Author :: getdata();
       cout << "Enter title, Price & No. of Pages:" << endl;
       cin >> title >> price >> pages;
     }
    void show()
     { Publisher :: show();
       Author :: show();
       cout << "Title:" << title << endl;
       cout << "Price:" << price << endl;
       cout << "No. of Pages:" << pages << endl;
     }
};
int main()
{ Book b;
  b.getdata();
  b.show();
  return 0;
}
```

(c) Hierarchical Inheritance:- When the traits of one class (Base class) may be inherited by more than one class (Derived classes) is called Hierarchical Inheritance

⇒ Here, A single Base class A is used for the derivation of two or more classes i.e. B, C and D.



(fig:- Hierarchical Inheritance)

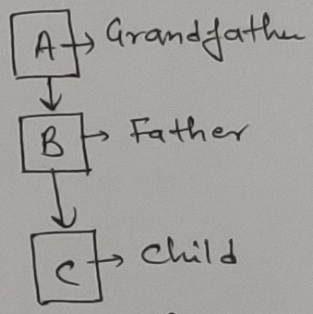
Example:-

```
#include<iostream>
using namespace std;
Class Account
{ int Acc_no; string cust_name;
  public;
  void getdata()
  { cout<<" Enter Account No. and Customer Name : "<<endl;
    cin>>Acc_no>>cust_name;
  }
  void show()
  { cout<<"Account Number:"<<Acc_no<<endl;
    cout<<"Customer Name :"<<cust_name<<endl;
  }
};
Class Sav_Acc : public Account
{ float roi;
  public:
  void getdata()
  { Account::getdata();
    cout<<"Enter Rate of Interest :"<<endl;
    cin>>roi;
  }
  void show()
  { cout<<"*** SAVING ACCOUNT ***"<<endl;
    Account::show();
    cout<<"Rate of Interest : "<<roi<<endl;
  }
};
Class Cur_Acc : public Account
{ float roi;
  public:
  void getdata()
  { Account::getdata();
    cout<<"Enter Rate of Interest:"<<endl;
    cin>>roi;
  }
  void show()
  { cout<<"*** CORRENT ACCOUNT ***"<<endl;
    Account::show();
```

```
    cout << "Rate of Interest :" << r01 << endl;
  }
};
int main()
  {
    Sav_Acc S;        // saving Account's object
    S.getdata();
    S.show();
    Cur_Acc C;        // Current Account's object
    C.getdata();
    C.show();
    return 0;
  }
```

(d) Multi-level Inheritance:- The mechanism of deriving a class from another derived class is known as multilevel inheritance. as shown in given figure.

Here, B is not only a derived class but also a base class for C. Further, C can be used as a base class for some another derived class.



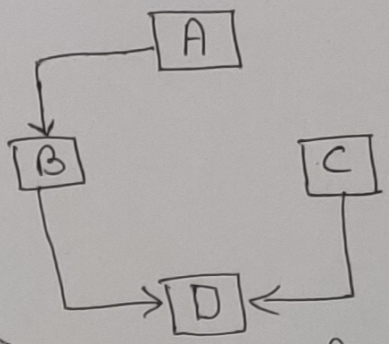
(fig:- Multilevel Inheritance)

Example:- Let's consider ~~perv~~ previous example of class Publisher for multi-level inheritance.

```
#include<iostream>
  using namespace std;
  class Publisher
   {
      string pname;
      string place;
    public:
     void getdata()
     { cout << "Enter name and place of publisher:" <<endl;
       cin >> pname >> place;
     }
    void show()
```

```
    {
      cout << "Publisher Name:" << pname << endl;
      cout << "Place:" << place << endl;
    }
};
  class Author : public Publisher
    {
      string Aname;
      public:
        void getdata()
        {
          Publisher::getdata();
          cout << "Enter Author Name:" << endl;
          cin >> Aname;
        }
       void show()
        {
          Publisher::show();
          cout << "Author Name:" << Aname << endl;
        }
    };
  Class Book : public Author
    {
        string title;
        float price;
        int pages;
      public:
        void getdata()
        {
          Author::getdata();
          cout << "Enter Book Title, Price & No. of Pages:" << endl;
          cin >> title >> price >> pages;
        }
        void show()
        {
          Author::show();
          cout << "Title:" << title << endl;
          cout << "Price:" << price << endl;
          cout << "No. of Pages:" << pages << endl;
        }
    };
int main() {
              Book b;
              b.getdata();
              b.show();
              return 0;
            }
```

(e) Hybrid Inheritance:- A combination of one or more types of inheritance is known as hybrid inheritance.



(fig:- Hybrid Inheritance)

⇒ Given figure, two types of inheritance is used. i.e. single and multiple inheritance

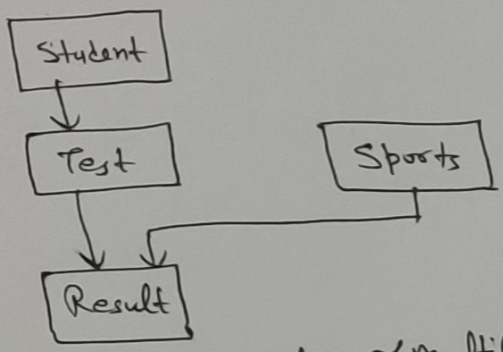
⇒ Here, class B is derived from class A so it is a single type of inheritance.

⇒ while class B acts as a base class for class D & it is derived from base classes B and C so it a multiple inheritance.

⇒ hence, combination of one or more types of inheritance is called as Hybrid Inheritance.

Example:- Let's consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Also assume, we have to give weightage for sports before finalizing result.

⇒ class Student stores the roll no., class Test stores the marks obtained in two subjects and class ~~Result~~ Result contains the total marks obtained in the test.



[fig:- Combination of multilevel & multiple Inheritance (Hybrid Inheritance)]

⇒ The class Result can inherit the details of marks obtained in test and roll no. of students through multilevel inheritance and the weightage for sports which is stored in a separate class called Sports; hence, result will have both the multilevel and multiple inheritances.

& its declaration would be —

```
Class Result : public Test, public Sports
{
  ---  // function body
};
```

where Test itself is a derived class from Student.

```
class Test : public Student
 {
   ——— // function body
 };
```

⇒ We can see the implementation of both multilevel and multiple inheritances in given program —

```
#include <iostream>
using namespace std;
class Student
 {
   protected:
     int roll_no;
     public:
      void getdata (int a)
      {
        roll_no = a;
      }
      void putdata ( )
      {
        cout << "Roll No: " << roll_no << "\n";
      }
 };
 class Test : public Student
  {  protected:
      float part1, part2;
     public:
      void getmarks (float x, float y)
       {   part1 = x;   part2 = y;
       }
      void putmarks ( )
       {
         cout << "Marks obtained :" << "\n"
              << "Part1 =" << part1 << "\n"
              << "Part2 =" << part2 << "\n";
       }
  };
 class Sports
  {  protected:
       float score;
```

```
public:
    void getscore(float s)
    {   score = s;
    }
    void putscore()
    {
        cout << "sports wt: " << score << "\n";
    }
};
Class Result : public Test ; public Sports
{   float total;
    public:
    void display();
};
void Result
void Result :: display()
{   total = part1 + part2 + score;
    putdata();
    putmarks();
    putscore();
    cout << "Total score:" << total << "\n";
}
int main()
{   Result st1;
    st1.getdata(125);
    st1.getmarks(33.7, 48.1);
    st1.getscore(7.1);
    st1.display();
    return 0;
}
```

output of this Program will be like —

```
Roll No: 125
Marks Obtained:
Part1 = 33.7
Part2 = 48.1
Sports Wt: 7.1
Total score = 88.9
```

Limitations :-

⇒ In inheritance, the base and inherited classes get tightly coupled, so their independent use is difficult.

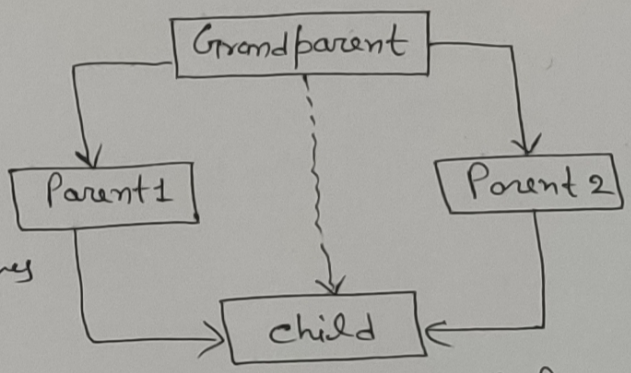
⇒ Wastage of memory and compiler overheads are the drawbacks of inheritance.

Virtual Base Classes :-

⇒ We have seen a situation which would require the use of both the multiple and multilevel inheritance.

⇒ Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance are involved.

⇒ This is shown in given figure. The "child" has two direct base classes "parent1" and "Parent2" which themselves have a common base class "Grandparent".



(fig:- Multipath Inheritance)

⇒ The "child" inherits the traits of "Grandparent" via two separate paths. It can also inherit directly as shown by the dotted line. The "Grandparent" is sometimes referred to as indirect base class.

⇒ Inheritance by the 'child' as shown in above figure. might face some problems. All the public and protected members of "grandparent" are inherited into 'child' twice, first via "parent1" and again via 'Parent2' This means 'child' would have duplicate sets of the members inherited from "Grandparent". It introduces ambiguity and should be avoided.

⇒ The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:

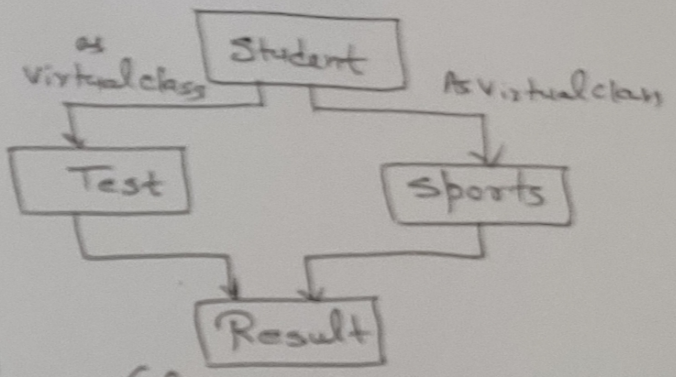
```
class G                          // Grandparent
{
  ----
  ----
};
class P1 : virtual public G      // Parent 1
{
  ----
  ----
};
class P2 : public virtual G      // Parent 2
{
  ----
  ----
};
class Child : public P1, public P2   // child
{ ----                           // only one copy of G will be
  ----                             inherited.
};
```

⇒ Here, the keywords "virtual" and "public" may be used in either order.

⇒ When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between virtual base class and a derived class.

Example :- Consider again the student results processing system & assume that the class sports derives the roll_no. from the class "student" Then the inheritance relationship will be as shown in figure —

⇒ In Given fegure, Class Student acts as a virtual class, hence it is free from or avoids ambiguity in student result procesing system.



(fig:- Virtual base class)

⇒ A Program to implement the concept of virtual base class is given below ——

```
#include <iostream>
using namespace std;
{
   protected:
      int roll_no;
      public:
      void get_no(int a)
       {
          roll_no = a;
       }
      void put_no()
       {
        cout <<"Roll no: " << roll_no << "\n";
       }
   };
   class Test: virtual public Student
    {  protected:
       float part1, part2;
       public:
       void get marks (float x, float y)
       {     part1 = x;  part2 = y;
       }
       void put marks( )
       {  cout << "Marks obtained :" << "\n"
                << "Part1" << part1 << "\n"
                << "Part 2" << part2 << "\n";
       }
    };
```

```
class Sports : public virtual Student
{
    protected:
        float score;
    public:
    void getscore(float s)
    {
        score = s;
    }
    void putscore()
    {
        cout << " sports wt :" << score << "\n";
    }
};
class Result : public test, public sports
{
    float total;
    public:
    void display();
};
void Result :: display()
{
    total = part1 + part2 + score;
    put_no();
    putmarks();
    putscore();
    cout << "Total score:" << total << endl;
}
int main()
{
    Result stud1;
    stud1.get_no(36);
    stud1.getmarks(15.7, 23.2);
    stud1.getscore(6.0);
    stud1.display();
    return 0;
}
```

⇒ here, only one copy of roll_no. variable is inherited by the result class from the student class. Thus, any changes made to roll_no. by any of the derived classes are made to this single instance. This ensures data integrity

Abstract Class :- An abstract class is one that is not used to create objects. It is designed only to act as a base class (to be inherited by other classes).

⇒ It is a design concept in program development and provides a base upon which other classes may be built

⇒ In real programming scenarios, the concept of abstract base classes holds great significance. They are deliberately used in a program for creating derived classes and are not meant for creating instance objects.

⇒ Let's take an example of an abstract base class vehicle which may be declared in a program for deriving LMV (light motor vehicle), HMV (heavy motor vehicle) and TW (two wheels) derived classes.

⇒ A pure virtual function spec() may set the specifications of LMV as per its four wheels and smaller engine and that of HMV as per its eight or more wheels and relatively larger engine.

⇒ Note that, a class can only be considered as an abstract class if it has at least one pure virtual function.

⇒ The general form of using abstract class is shown below —

```
class vehicle                    // abstract base class
{ private:
    data-type d1;
    data-type d2;
  public:
    virtual void spec()=0;       // pure virtual fun^n
};
class LMV : public vehicle
{   public:
```

```
    void spec()
    {
    //LMV definition of spec function
    }
};
class HMV : public vehicle
{
    public:
        void spec()
        {
        // HMV definition of spec function
        }
};
class TW: public vehicle
{
    public:
        void spec()
        {
        // TW definition of spec fun^n
        }
};
```

Inline Functions:- One of the objectives of using functions in a program is to save some memory space which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function.

⇒ When a function is small, a substantial percentage of execution time may be spent in such overheads.

⇒ C++ has a solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called Inline function.

⇒ An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion).

⇒ General syntax of this function is—

```
inline function-header
{
  //function body
}
```

Ex-
```
inline int rectangle (int l, int b)
{
  return (l*b);
}
```

This inline function can be invoked by statement like –

```
b = rectangle(3,4);
c = rectangle(2+1, 3+2);
```

⇒ It is easy to make a function inline. All we need to do is to prefix the keyword inline to the function definition.

⇒ All inline functions must be defined before they are called.

⇒ Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

⇒ Some of the situations where inline expansion may not work are –

a) For functions returning values, if a loop, a switch, or a goto exists.

b) For functions not returning values, if a return statement exists

c) If functions contain static variables.

d) If inline functions are recursive.

⇒ Given implementation shows the use of inline functions.–

```
# include <iostream>
    using namespace std;
```

```
inline int add (int x, int y)
  {
     return (x+y);
  }
inline float mul (float p, float q)
    { return (p*q);
    }
int main()
{
    add    int a = 10;
           int b = 15;

    cout << add (a, b) << "\n";
    cout << mul (a, b) << "\n";

    return 0;
}
```

⇒ o/p of this program would be –

25
150

⇒ Note that, speed benefits of inline functions diminish as the function grows in size.

Friend Functions :-

⇒ The functions that are declared with the keyword friend are known as friend functions.

⇒ The function declaration should be preceded by the keyword friend as shown given below –

```
Class Demo
  {
    ==========
    public:
    ==========
      friend void abc();    // declaration
  };
```

⇒ A function can be declared as a friend in any number of classes.

⇒ A friend function, although not a member function, has full access rights to the private member of the class.

⇒ A friend function have following characteristics —

a) It is not in the scope of the class to which it has been declared as friend.

b) Since it is not in the scope of the class, it can not be called using the object of that class.

c) It can be invoked like a normal function without the help of any object.

d) Unlike member functions, it can not access the member names directly and has to use an object name and dot membership operator with each member name.

e) It can be declared either in public or the private part of a class without affecting its meaning.

f) Usually, it has the objects as arguments.

⇒ Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below —

```
class X
{
  ==
  int fun(); // MF of X
  ==
};
class Y
  {
  ==
  friend int X :: fun();
  ==
};  // Here fun() is a member
    of class X and a friend
    of class Y.
```

⇒ We can also declare all the member functions of one class as the friend functions of another class. Hence the class is called a friend class.

```
class Z
  {
    ==
    friend X;   // "all member functions of X
  };                are friends to z class."
```

Ex:- Given program shows the use of a friend function -

```
#include <iostream>
  using namespace std;
  class sample
   {
      int a, b;
      public:
      void setvalue ()
      {a = 25; b = 30;}
      friend float mean (sample s);
   };
  float mean (sample s)
   {
     return float (s.a + s.b)/2.0;
   }
  int main ()
  {
     sample X;
     X.setvalue();
     cout << "mean value = " << mean(X) << endl;
     return 0;
  }
```

The o/p of this program would be -

mean value = 32.5