2nd Class
Object Oriented Programming (C++)

Lecture Notes on
Structures, Classes & Objects

By
Dr. Naseer Ali Hussein

Structures

A structure is a collection of simple variables. The variables in a structure can be of different types: Some can be int, some can be float, and so on. (This is unlike the array, which we'll meet later, in which all the variables must be the same type.) The data items in a structure are called the *members* of the structure. However, for C++ programmers, structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. A structure (as typically used) is a collection of data, while a class is a collection of both data and functions. So by learning about structures we will be paving the way for an understanding of classes and objects. Structures in C++ (and C) serve a similar purpose to *records* in some other languages such as Pascal.

A Simple Structure

Let's start off with a structure that contains three variables: two integers and a floating-point number.

```
#include<iostream.h>
#include<conio.h>
 struct database
 {
  int id_number;
  int age;
  float salary;
};
 int main()
{
  database employee;

  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;

  cout <<"age is = "<<employee.age<<endl;
  cout <<"id_number is = "<<employee.id_number<<endl;
  cout <<"salary is = "<<employee.salary<<endl;
  getch ();
}
```

Overview of OOP

Bjarne Stroustrup at Bell Labs developed C++ during 1985. The term C++ was first used in 1983. Prior to 1983, Stroustrup added features to C programming language and formed what he called "C with Classes". In addition to the efficiency and portability of C, C++ provides number of new features. C++ programming language is basically an extension of C programming language.

The fashion of the 1990s is most definitely object-oriented programming.

Basic concepts of Object-Oriented Programming:
It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

1. Classes
2. Objects
3. Encapsulation and Data Hiding.
4. Data Abstraction
5. Inheritance and Reuse.
6. Polymorphism.

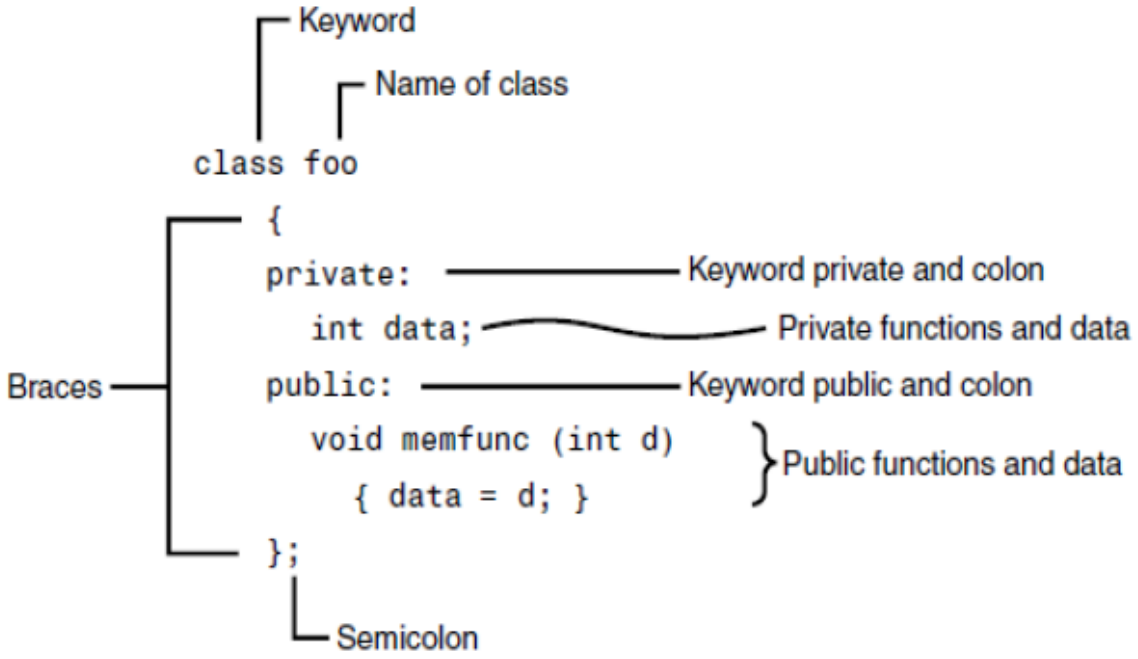
1. Class Definition:
Class is a keyword, whose functionality is similar to that of the **struct** keyword, but with the possibility of including functions as members, instead of only data.
Classes are collections of variables and functions that operate on those variables. The variables in a class definition are called data members, and the functions are called member functions.

A **class** specification has two parts:

- Class declaration: It describes the type and scope it its members. The class declaration is similar to a **struct** declaration
- Class function definitions: It describes how the class functions are implemented.

The variables declared inside the class are known as ***data members*** and the functions are known as ***member functions***.

A typical class declaration would look like:



Class members fall under one of three different access permission categories:

- **Public** members are accessible by all class users.
- **Private** members are only accessible by the class members. By default, the members of a class are private.
- **Protected** members are only accessible by the class members and the members of a derived class.

General form of class declaration:

```
class class-name
{
        public:
          public-data-members;
          public-functions;

        private:
          private-data-members;
          private -functions;

        protected:
          protected-data-members;
          protected -functions;
};
```

Note:

Only the member functions can have access to the **private** data members and private functions. However, the **public** members (both functions and data) can be accessed from outside the class.

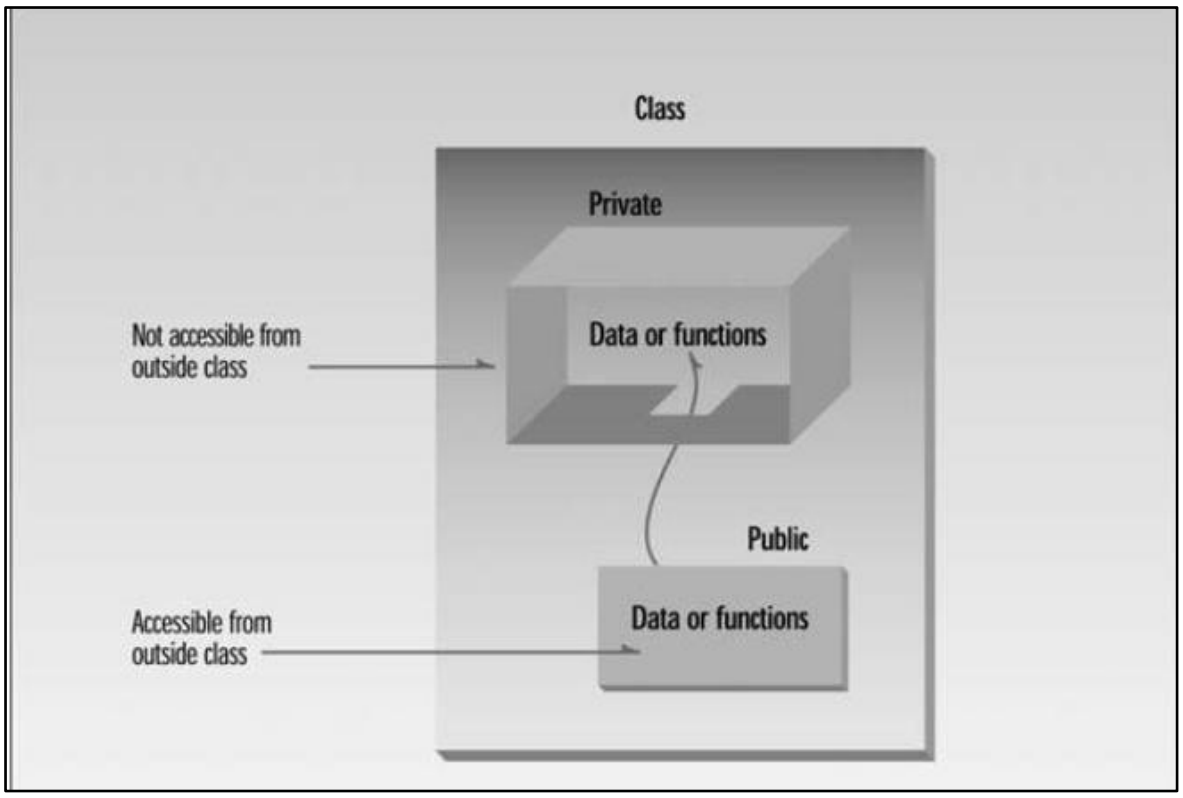


Figure 3: Data binding in classes

2. Objects

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. Objects are the basic run-time entities in an object-oriented system. They may be present a person, a place, a bank account, or any item that the program has to handle.

In fact, objects are variables of the type ***class***, once a class has been defined, we can create any number of objects belong to that class. Each object is associated with the data of type class with which they are created.

A Simple Class Example:

```
class item
{
int number;                 // variables declaration
float cost;                 //private by default
```

```
public:
void getdata(int a, float b)    //functions declaration
void putdata(void)              //using prototype
};
```

Creating Object

Once a class has been declared, we can create variables of that type by using the class name (like other built-in- type variable). For example:

item x; // memory for x is created

creates a variable x of type item. The class variables are known as objects. Therefore x is called an object of type item.

We may also define more than object in one statement. Example:

item x,y,z;

The necessary memory space is allocated to an object at this stage.

Accessing Class members

The private data of a class can be accessed only through the member functions of that class. The **main ( )** cannot contain statements that access number and cost directly. The following is the format for calling a member function:

Object-name. function-name(actual-arguments);

For example, the function call statement

x.getdata(100,75.5);

is valid and assigns the value 100 to number and 75.5 to cost of the object x by implementing the **getdata()** function. Remember, a member function can be invoked only by using an object (of the same class). The statement

x.number=100; is also illegal.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
int x;
int y;
public:
int z;
};
…….
xyz p;
```

p.x=0; // error, x is private
p.z=10; // OK, z is public
…….
Note that the use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

Example 1:

Write a C++ program to modeling the Rectangle class.

```
# include <iostream.h>

class Rectangle
{
    public:
        int length , width;
        int area( )
        {
           return length * width;
        }
};


int main( )
{
  Rectangle my_rectangle;

  my_rectangle.length = 6;
  my_rectangle.width = 7;
  cout<< my_rectangle.area( );

  return 0;
}
```

Defining Member Functions

Member functions can be defined in two places:
• Inside the class definition.
• Outside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle different in the way the function header is defined.

➢ Inside the Class Definition

The method of defining a member function is to replace the function declaration by the actual function definition inside the class. Normally, only small functions are defined inside the class definition. For example:

Example 2:A Simple Class

```
#include <iostream>
using namespace std;
class smallobj     //define a class
{
private:
int  somedata; //class data
public:
void  setdata(int d)            //member function to set data
{
somedata = d;
 }
void showdata()               //member function to display data
{ cout << "Data is " << somedata << endl; }
};
int main()
{
smallobj s1, s2;            //define two objects of class smallobj
s1.setdata(1066);      //call member function to set data
s2.setdata(1776);
s1.showdata();          //call member function to display data
s2.showdata();
return 0;
}
```

Example 3: write an oo program to define the coordinate of point and change the values of point.

```
#include<iostream.h>
class point {
        int xval , yval;
public:
   void setpt(int x , int y)
{
        xval=x;
        yval=y;
}
   void offsetpt(int x , int y)
{
        xval+=x;
        yval+=y;
              cout<<xval<<yval;

}
};
void main()
{
 point pt;
pt.setpt(10,20);
pt.offsetpt(2,2);
}
```

- **Outside the Class Definition**

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. The general form of a member function definition is:

return-type class-name* : : *function-name (argument declaration)
{
Function body
}

The membership label *class-name* **: :** tells the compiler that the function *function-name* belongs to the class *class-name.* That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol **: :** is called the ***scope resolution operator.***

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.
For instance, consider the member functions **getdata( )** and **putdata( ).** They may be coded as follows:

```
class item
{
int number;
float cost;

public:
void getdata(int a, float);              // function declaration
};
void item: : getdata(int a, float b)    // function definition
{
number = a;
cost = b;
}
```

```
//////////////////////[ class implementation ]////////////////////////////////
// ..... include section.....
#include <iostream.h>

// ..... class declaration .....
Class item              // class declaration
{
      int number;    // private by default
      float cost;
 public:
      void getdata(int a, float b); // prototype declaration
      void putdata(void)
      {
            cout <<"number :" << number << "\n";
            cout <<"coust     :"<<coust<<"\n";
       }
};
// ..... member function definition .....
void item::getdata(int a, float b)  // use membership label
{
     number = a;   // private variables
     cost    = b;     // directly used
}
// ..... Main program .....
int main (void)
{
     item x;                                  // create object x
     cout << "\n object  x "  << "\n";
     x.getdata(100, 299.95);                  // call member function
     x.putdata();

     item y;                                  // create object y
     cout << "\n object  y "  << "\n";
     y.getdata(200, 175.50);                  // call member function
     x.putdata();
     return (0);
}
//////////////////////////< program 5.1>////////////////////////////////
```

The program shows that:

- The member functions can have direct access to private data items.
- The member function putdata( ) has been defined inside the class and therefore behaves like an inline function.
- The program creates two object x and y in two different statements. This can be combined in one statement.

3. Encapsulation and Data hiding

The binding of data and functions together into a single class-type variable is referred to as ***encapsulation.***

Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally. C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.

4. Data Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract ***attributes*** such as size, weight and cost, and functions to operate on these attributes.

The attributes are sometimes called ***data members*** because they hold information. The functions that operate on these data are sometimes called ***methods or member functions***.

5. Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. In OOP, the concept of Inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

6. Polymorphism

Polymorphism, in a Greek term, means the ability to take more than one form (Poly means many, and morph means form). Polymorphism refers to the same name taking many forms. Using a single function name to perform different types of tasks is known as ***function overloading***.

- **Nesting of Member Functions**

We discussed that a member function of a class can be called by an object of that class using a *dot operator*. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as ***nesting of member functions***. The following program illustrates the concept of nesting member functions.

```
#include<iostream.h>
class set
{
int m, n;
public:
void input (void);
void display(void);
int largest (void);
};
int set :: largest (void)
{
if (m >=n)
return (m);
else
return (n);
}
void set :: input (void)
{
cout <<"input values of m and n"<<"\n";
cin >>m>>n;
}
void set :: display (void)
{
cout <<"largest value="<<largest()<<"\n";
}
int main ()
{
 set A;
 A.input();
 A.display();
 return 0;
 }
```

The above program shows the greatest number among two numbers.
Output:

```
input the values of m and n
4
8
the largest value is=8
```

- **Private Member Function**

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

Class Sample
{
int m;
void read (void);
public:
void update (void);
void write (void);
};

If s1 is an object of sample, then
S1.read (); *//wont work; object cannot access because its private member function*

is illegal, however, the function ***read ()*** can be called by the function ***update ()*** to update the value of m.

void sample :: update (void)
{
read ();
}

- **Static Data Members**

A static member variable has certain special characteristics. These are:

1. It is initialized to zero when the first object of its class is created.
2. Only one copy of that member is created for the entire class and is shared by all the objects of that class. No matter how many objects are created.
3. It is visible only within the class, but its lifetime in the entire program.

A static data member can be used as a counter that records the occurrences of all the objects. The following program illustrates the use of static data member.

```
#include <iostream.h>
class item
{
static int count;
int number;
public:
void getdata (int a)
{
number=a;
count ++;
}
void getcount(void)
{
cout <<"the count is :";
cout <<count<<endl;
}
};
int item :: count;
int main ()
{
item a, b, c;
a.getcount ();
b.getcount ();
c.getcount ();

a.getdata (10);
b.getdata (20);
c.getdata (30);
cout <<"After reading data"<<"\n";
a.getcount ();
b.getcount ();
c.getcount ();
return 0;
}
```

The output of the program would be:

```
the count is :0
the count is :0
the count is :0
After reading data
the count is :3
the count is :3
the count is :3
```

- **Static Member Functions**

  Like static member variable, we can also have static member functions. A member function that is declared **static** has the following properties:

  1. A static function can have access to only other static members (functions or variables) declared in the same class.
  2. A static member function can be called using the class name (instead of its objects) as follows:

     class-name :: function-name;

```
#include <iostream.h>
class test
{
int code;
static int count;
public:
void setcode(void)
{
code = ++count;
}
void showcode(void)
{
cout <<"object number :"<<code<<"\n";
}
static void showcount(void)
{
cout<<"count:"<<count<<"\n";
}
};
int test ::count;
int main ()
{
test t1, t2;
t1.setcode();
t2.setcode();
test ::showcount();
test t3;
t3.setcode();
test ::showcount();
t1.showcode();
t2.showcode();
t3.setcode();
return 0;
}
```

The output of the program would be:

```
count:2
count:3
bject number :1
bject number :2
bject number :3
```

Note:

- The statement ***(code = ++count)*** is executed whenever the function ***setcode()*** is invoked and the current value of ***count*** is assigned to ***code***. Since each object has its own copy of ***code,*** the value contained in ***code*** represents a unique number of its object.

- The static function ***showcount ()*** displays the number of objects created till that moment. A count of number of objects created is maintained by the **static** variable count.

- The function **showcode ()** displays the code number of each object.

Remember, the following function definition will not work:

```
Static void showcount ()
{
  cout<<code;       // code is not static
}
```

1. FRIEND FUNCTIONS

- **What is a Friend Function?**
  A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

- **Need for Friend Function**
  When a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. We may have situations where we would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful.

- **How to define and use Friend Function**
  The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. To make an outside function "friendly" to class, we have to simply declare this function as a ***friend*** of the class as shown below:

```
class ABC
{
      ........
      ........
public:
      ........
      ........
      friend void xyz(void) //declaration
};
```

- **A friend function has certain special characteristics**

✓ The keyword ***friend*** is placed only in the function declaration of the friend function and not in the function definition.
✓ It is possible to declare a function as ***friend*** in any number of classes.
✓ A friend function, even though it is not a member function, would have the rights to access the private members of the class.
✓ It is possible to declare the ***friend*** function as either private or public.
✓ The function can be invoked without the use of an object. The ***friend*** function has its argument as objects.
✓ Unlike member functions, it cannot access the member names directly and has to use an object name and dot operator with each member name. (e.g. A.x).

The following program illustrates the mean value of any two numbers by using friend function.

```
#include <iostream.h>
class TEST
{
  int A,B;
  friend float mean (TEST S);
public:
  // Member function definition
  void setvalue()
{
cout <<"eneter the values of A and B"<<"\n";
cin>>A>>B;
}
};
// function mean()is not a member function of any class.
float mean (TEST S)
{
 //Because mean() is a friend of TEST,
 // it can directly access any member of this class

return (S.A+ S.B)/2.0;
}

int main( )
{
TEST S;
   S.setvalue();
   //call friend function mean () to print the mean value
 cout<<"mean value ="<< mean(S)<<"\n";
 return 0;
}
```

The output of the program would be:

```
eneter the values of A and B
10
2
mean value =6
```

The program below demonstrates how friend functions work as a bridge between the classes. Note that the function ***max ( )*** has arguments from both ***XYZ and ABC***. When the function ***max ( )*** is declared as a friend in ***XYZ***, for the first time, the compiler will not acknowledge the presence of ***ABC*** under its name is declared in the beginning as ***class ABC***;

```
#include <iostream.h>
class ABC;
class XYZ
{
int x;
public:
void setvalue (int i)
{
x=i;
}
 friend void max (XYZ m, ABC n);
};

class ABC
{
int a;
public:
void setvalue (int i)
{
a=i;
}
friend void max(XYZ m, ABC n);
 };
void max (XYZ m, ABC n)
{
if (m.x >= n.a)
cout <<m.x;
else
cout <<n.a;
}
int main ()
{
ABC abc;
abc.setvalue (10);
XYZ xyz;
xyz.setvalue (20);
max(xyz, abc);
return 0;
 }
```

2. FRIEND CLASS

It is also possible to make an entire class a friend of another class. This gives complete access to all its data and methods including private and protected data and methods to the friend class member methods. Friendship is one way only, which means if A declares B as its friend it does

NOT mean that A can access private data of B. It only means that B can access all data of A. The class is called a friend class. This can be specified as follows:

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

The following program illustrates the maximum value of any two numbers by using friend class.

```
#include <iostream.h>
class TEST
{
private:
int x,y;
public:
void input()
{
cout<<"enter the value of x and y"<<"\n";
cin>>x>>y;
}
friend class display; //all member functions of display are friend to point
};
class display
 {
 public:
 void max (TEST p)
 {
 if (p.x >=p.y)
 cout<<"the largest value is x="<<p.x<<endl;
 else
 cout<<"the largest value is y="<<p.y<<endl;
 }
 };


 int main ()
 {
 TEST p;
 p.input();
 display S;
 S.max(p);
 return 0;
 }
```

The output of the program would be:

```
enter the value of x and y
10
6
the largest value is x=10
```

The below program represents a friend class to class point:

```
#include <iostream.h>
class point
{
int xval, yval;
public :
void setpt (int x, int y)
{
xval=x+2;
yval=y+3;
}
friend class display; //all member functions of display are friend to point
};
class display
 {
 public:
 void showdata (point p)
 {
 cout<<"xval="<<p.xval<<endl<<"yval="<<p.yval<<endl;
 }
 };
 int main ()
 {
 point p;
 p.setpt (10,2);
 display x;
 x.showdata (p);
 return 0;
 }
```

The output of the program would be:

xval=12
yval=5

Note: The ***display class*** is a friend of ***point class***, any of ***Display's*** members that use a ***point*** class object can access the private members of ***point*** directly.

➢ **What is the different between Friend Function and Friend Class?**

- ***Friend function***

✓ The friend keyword is used for declaration.
✓ While writing definition of function, the friend keyword is not required.
✓ Through a friend function, we can allow outside functions to access the class members.

- ***Friend class***

✓ For the declaration of a friend class, the friend keyword is used: friend class a;
✓ While writing a class, the friend keyword is not required.

- ✓ With a friend class we can access the members of one class into another.

- **What is different between friend function and member function in C++?**

- ✓ A member function is the one which is defined inside a class and is a member of the class. It may either be a public private or protected function.

- ✓ We use friend function in the case when we want to one class to communicate with other class. For this we need to declare that friend function in both the classes and define that friend function outside the associated classes. The friend function is always capable to access all the members of the associated classes.

- ✓ The major difference is that a friend function is called like f(x), while a member function is called like x.f(). Thus the ability to choose between member functions (x.f()) and friend functions (f(x)) allows a designer to select the syntax that is deemed most readable, which lowers maintenance costs.

Class Constructors and Destructors

3. INTRODUCTION

A constructor is a special member function that is executed automatically whenever we create new objects of that class. It is special because its name is the same name as the class name. The main purpose of a class constructor is to perform any initializations related to the class instances via passing of some parameter values as initial values and allocate proper memory locations for that object.

Characteristics of Constructors

- They should be declared in the public section
- They are automatically invoked and have no return types, neither can they return values
- They cannot be virtual nor can we refer to their addresses & they can have default arguments
- They make implicit calls to new and delete

General Syntax of declaration and definition of constructor as follows:

```
class interger
{
int m, n;
public:
integer ();                 //constructor declared
……..
};
integer :: integer ()       //constructor defined
{
……….
}
```

Types of constructors

A. **Default constructor:-** Default Constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we creates the object of class but Remember name of Constructor is same as name of class and Constructor never declared with the help of return type. When the class contains the default constructor like in above example ***integer ()***. It is guaranteed that an object created by the class will be initialized automatically. For example,

```
integer  int1                    // object int1 created
```

Example 1: WAP to add two numbers by using default constructor.

```
#include <iostream.h>
class Add
{
int m, n;
public:
Add()     //default constructor
{
cout<<"enter the values of m and n"<<endl;
cin>>m>>n;
cout <<"The values of m + n ="<<m+n<<endl;
}
};
int main ()
{
Add A;
return 0;
}
```

B. **Parameterized Constructor: -** This is another type of constructor which has some Arguments and same name as class name but it uses some arguments. For this type we have to create object of class by passing some arguments at the time of creating object with the name of class. When we pass some Arguments to the constructor then this will automatically pass the arguments to the constructor and the values will retrieve by the respective data members of the class.

class ***interger***
{
int m, n;
public:
integer *(int x, int y);* ***// Parameterized constructor declared***
…….
};
integer :: ***integer*** *(int x, int y)* ***// Parameterized constructor defined***
{
m=x;
n=y;}

When a constructor has been parameterized, the object declaration statement such as
integer int1 //may not work.
We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

1. By calling the constructor explicitly.
   integer int1= integer (0,100); //explicit call
   the statement creates an integer object called ***int1*** and passes the values 0 and 100 to it.

2. By calling the constructor implicitly.
   integer int1 (0,100); ***//implicit call***
   This method sometimes called the shorthand method because is used very often as it is shorter, looks better and is easy to implement.

Note:

1. The parameters of a constructor can be of any type except that of the class to which it belongs. For example:

```
class A
{
int m, n;
public:
A (A);     // is illegal
};
```

2. A constructor can accept a reference to its own class as a parameter. Thus, the statement

```
class A
{
int m, n;
public:
A (A&);     // is legal which can called (copy constructor)
};
```

Example 2: WAP to find the area of the rectangle by using parameterized constructor.

```
#include <iostream.h>
class Rectangle
{
int length, width;
public:
Rectangle (int x, int y)
{
length=x;
width=y;
}
int area ()
{
return length * width;
}
};
int main ()
{
Rectangle R (6,2);                    //implicit call
//Rectangle R= Rectangle (6,2);  //explicit call
cout <<"The area of the Rectangle is "<<R.area()<<endl;
return 0;
}
```

C. **Copy Constructor**:- A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

class ***interger***

{
int m, n;
public:
integer** (**integer** & i) // **Copy constructor
{
m=i.m; n=i.n;}
};

When a copy constructor has been declared and defined, the object declaration statement such as:
integer int1(int2);

Would define the object ***int1*** and the same time initialize it to the values of ***int2***. Another of this statement is

integer int1= int2;

Note:

- The process of initializing through a copy constructor is called copy initialization
- A copy constructor takes a reference to an object of the same class as itself.
- ***int1=int2*** will copy member by member but does not call the copy constructor.
- We cannot pass by value to a copy constructor.

Since we discussed that there are three kinds of constructors (Default, Parameterized, and Copy Constructor). We can have multiple constructors in one class. We learned that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in class, we say that the constructor is overloaded.

Example3: WAP to display the student's ID number and marks by using multiple constructors. ***Assume that the first student has any ID number with any marks, the second student has ID number 5 and the marks 78 and the third student has the same ID number and marks of second student.***

```
#include<iostream.h>
 class student
 {
   private :
   int Idnumber, marks;
   public:
   student() //default constructor 1
   {
   cout<<"Enter The Student ID Number : ";
   cin>>Idnumber;
   cout<<"Enter The Student Marks : ";
   cin>>marks;
    }
   student(int r, int m) //parameterized constructor 2
   {
     Idnumber=r;
     marks=m;
   }
   student(student &t) //copy constructor 3
   {
      Idnumber=t.Idnumber;
      marks=t.marks;
   }
   void showdata() // member function to show data
   {
```

```
    cout<<"\nThe ID Nmber is:"<<Idnumber<<"\nThe Marks is:"<<marks;
}
};
 int main()
   {
    student st1;          //defalut constructor invoked
    student st2(5,78);  //parmeterized constructor invoked implicitly
    student st3(st2);  //copy constructor invoked
    st1.showdata();     //display data members of object st1
    st2.showdata();    //display data members of object st2
    st3.showdata();    //display data members of object st3
  return 0;
   }
```

4. DESTRUCTOR

A destructor is used to clean up the object just before it is destroyed. A destructor always has the same name as the class itself, but is preceded with a ~ symbol. Unlike constructors, a class may have at most one destructor. A destructor never takes any arguments and has no explicit return type.

~ student () { } // destructor declaration
Student :: ~student () //destructor definition outside the class { }

Destructors have specific naming rules:
1) The destructor must have the same name as the class, preceded by a tilde (~).
2) The destructor cannot take arguments.
3) The destructor has no return type.

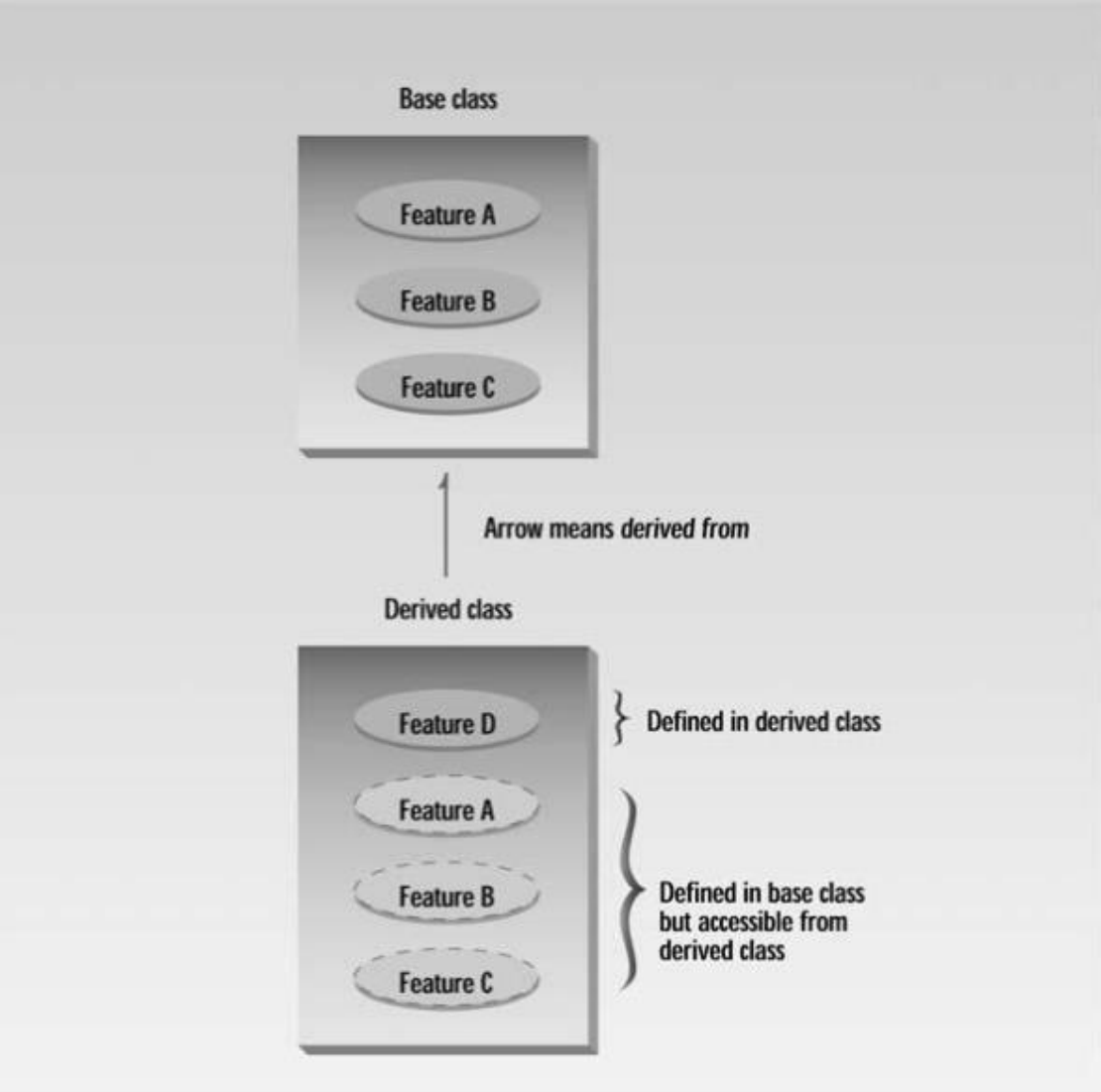
Example 4: WAP to represent the destructor in rectangle class.

```
#include <iostream.h>
class Rectangle
{
int length, width;
public:
Rectangle (int x, int y)
{
length=x;
width=y;
}
~ Rectangle ()
{
cout <<"destructor delete data"<<endl;
}
int area ()
{
return length * width;
}
};
int main ()
{
Rectangle R (6,2);                    //implicit call
cout <<"The area of the Rectangle is "<<R.area()<<endl;
return 0;
}
```

Inheritance: Extending Classes

5. INTRODUCTION

Inheritance is probably the most powerful feature of object-oriented programming after classes themselves. 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.



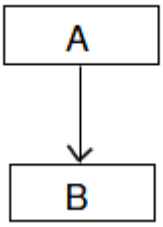
Inheritance is an essential part of OOP. Its big payoff is that it permits code reusability. Once a base class is written and debugged, it need not be touched again, but, using inheritance can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability.

An important result of reusability is the ease of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

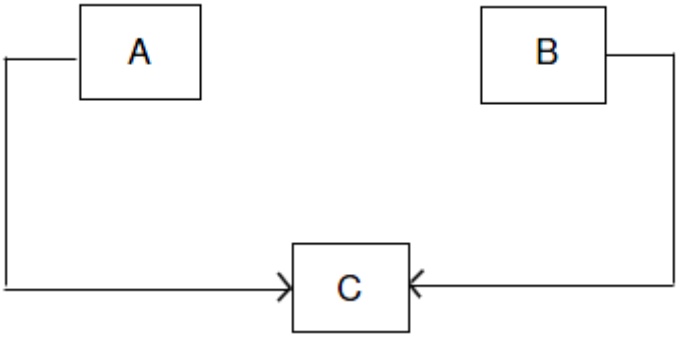
Different Forms of Inheritance

The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and new one is called the derived class. There are various forms of inheritance.

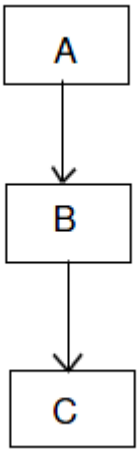
- **Single inheritance**: A derived class with only one base class is called single inheritance.



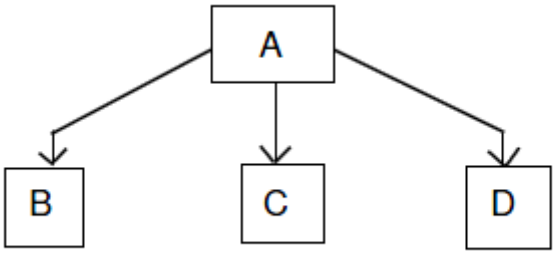
- **Multiple inheritance**: A derived class with several base classes is called multiple inheritance.



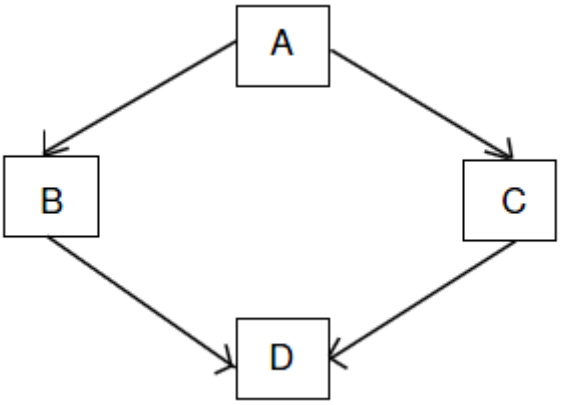
- **Multilevel inheritance**: The mechanism of deriving a class from another derived class is called multilevel inheritance.



- **Hierarchical inheritance:** One class may be inherited by more than one class. This process is known as hierarchical inheritance.



❖ **Hybrid inheritance**: It is a combination of hierarchical and multiple inheritance.



Defining Derived Class

A derived class is defined by specifying its relationship with the base class using visibility mode.

The general form of defining a derived class is:

```
class  Derived-class : visibility-mode Base-class
{
            // members of derived class
};
```

The colon indicates that the derived_class is derived (inherits some property) from base_class.

The visibility-mode can be either private or public or protected. If no visibility mode is specified, then by default the visibility mode is considered as ***private.***

Examples:

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

Important Notes:

1. When a base class is ***public*** inherited derived class, public members of the base class become '***public members'*** of the derived class and therefore they are accessible to the objects of the derived class.

2. When a base class is ***privately*** inherited by a derived class ,'public members of the base class become '***private members'*** of the derived class and therefore the public members of the base class can only be accessed by the member function of the derived class. They are ***inaccessible*** to the objects of the derived class.

3. In both cases, the private members of a base class are not inherited to the derived class.

4. In inheritance, some of the base class data elements and members functions are "inherited "into the derived class, we can add our own data and members functions and thus extend the functionality of the base class.

A. Single Inheritance:

Let us consider a simple example to illustrates inheritance. The following program shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public functions. The class D contains one private data member and two public member functions.

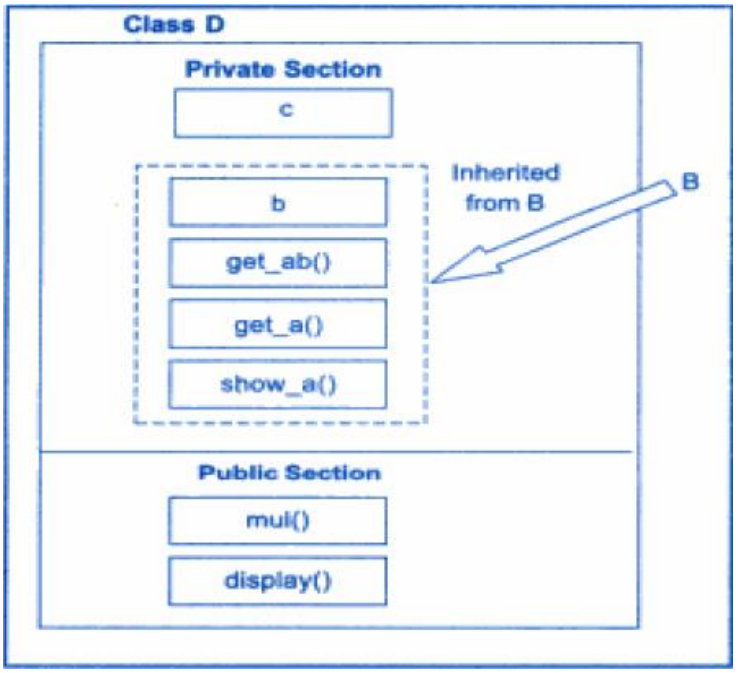
Example 1 (public Derivation)

```
#include <iostream.h>
class B
{ int a ; // private ; not inheritable
public :
int b ; // public ; ready for inheritance
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)
{
 cout<<"enter the values of a and b"<<"\n";
 cin>>a>>b;
}
int B ::get_a ()
{
return a ;
}
void B::show_a ()
{
cout<<"The value of a= " <<a<< "\n";
}
void D::mul()
{
c= b* get_a();
}
void D::display ()
{
 cout<<"The value of a = "<<get_a()<<"\n";
 cout<<"The value of b = "<< b <<" \n";
 cout<<"The value of c = "<< c <<"\n \n" ;
}
int main()
{
D d ;
d.get_ab(); // ok ,public
d.mul();
d.show_a(); // ok, public
d.display() ;
 return 0 ;
 }
```

The output of the program is:

```
enter the values of a and b
10
2
 The value of a= 10
 The value of a = 10
 The value of b = 2
 The value of c = 20

 The value of a = 10
 The value of b = 20
 The value of c = 200
```



Example 2 (Private Derivation)

```
#include <iostream.h>
class B
{
int a;      // private ; not inheritable
public :
int b ;      // public ; ready for inheritance
void get_ab();
int get_a(void);
void show_a(void);
};
class D : private B       // Private derivation
{
int c ;
public :
void mul(void);
void display(void);
};
void B ::get_ab(void)
{
 cout<<"enter the values of a and b"<<"\n";
 cin>>a>>b;
}
int B ::get_a()
{
return a;
}
void B :: show_a()
{
cout<<" The value of a= "<<a<<"\n";
}
void D :: mul()
{
get_ab();
c= b* get_a();
}
void D :: display ()
{
show_a();
 cout<<" The value of a = "<<get_a()<<"\n";
 cout<<" The value of b = "<< b <<" \n";
 cout<<" The value of c = "<< c <<"\n \n" ;
}
int main()
{
 D d ;
   d.mul();
   d.display();
   return 0 ;
 }
```

❖ Access Members with Different Visibility Modes

As we know in C++, there are three access modifiers namely, public, private and protected. The private members of the base class cannot inherit therefore it is not available for the derived class directly. The protected members of the base class can accessible with in its class and any class immediately derived from it. It cannot be accessed by any methods that are outside of these two classes i.e. methods do not belongs to these two classes.

A class can now use all three visibility modifiers as shown below:

Class A
{
Private *:*
// members visible only to this class member functions
Protected:
// members visible to this class member functions and also for its derived
// class member functions.
Public:
// visible to all functions in the program.
};

- When a protected member is inherited in public mode, it becomes protected in its derived class therefore this members can accessible by member functions of the derived class.

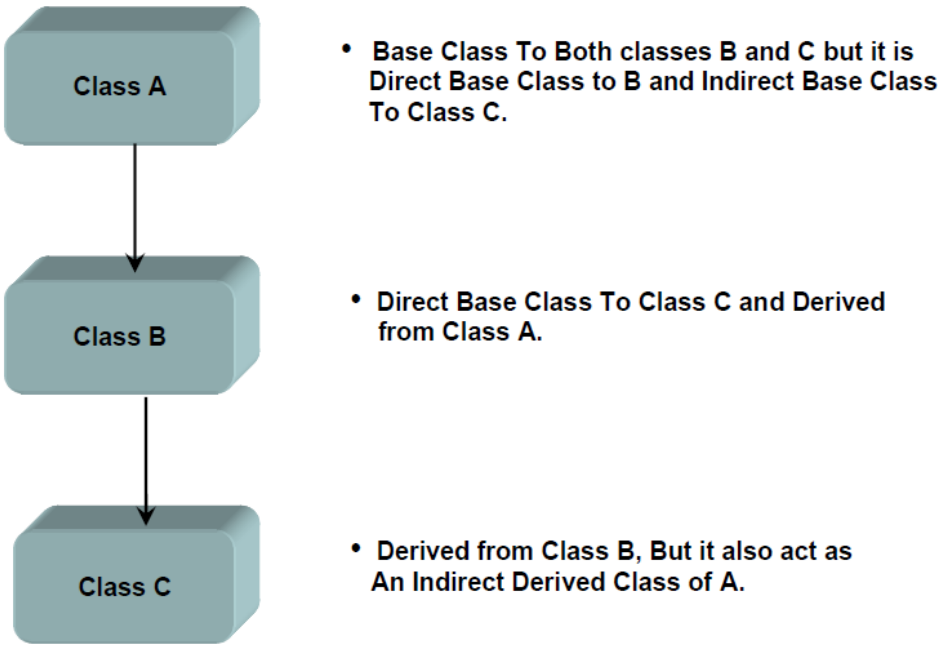
- When a protected member is inherited in private mode, it becomes private in its derived class therefore these members can accessible only by this class member functions, it is not available for further inheritance since private members cannot inherited.

The following table illustrates how visibility of base class members will undergo modifications in all the three kinds of derivation.

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

2. MULTILEVEL INHERITANCE

When you define more than two levels of inheritance (in the form of a chain of classes), it would be generally referred to as multi-level inheritance. In the case of multi-level inheritance, all the members of all super classes would be automatically available within the sub class.



Now we see how multi-Level inheritance should implement through this syntax.

Syntax:

```
Class base_class_name1
{
// List of members
};
Class derived_base_name2 : <visibility mode> base_class_name1
{
// List of members
};
Class derived : <visibility mode> derived_base_name2
{
// List of members
};
```

As you seen the syntax above clearly shows how to implement the multi-Level inheritance through programming.

Example 1:

Assume that the test results of a batch of students are stored in three different classes. Class A stores the roll number, class B stores the marks obtained in two subjects and class C contains the total marks obtained in the test. The class C can inherit the details of the marks obtained in the test and the roll number of students through multi-level inheritance.

```
#include<iostream.h>
class A
{
public:
int roll_No;
void getrollno (int r)
{
roll_No=r;
}
void showrollno ()
{
cout <<"student roll number: "<<roll_No<<"\n";
}
};
class B: public A
{
public:
int math, cs;
void getmak (int x, int y)
{
math=x;
cs =y;
}
void showmarks ()
{
cout<<"The marks in mathematics = "<< math<<"\n";
cout<<"The marks in computer science = "<<cs<<"\n";
}
};
class C: public B
{
public:
int total;
void gettotal()
{
total = math + cs;
}
void showtotal()
```

```
{
cout<<"total marks= "<<total;
}
};
void main ()
{
C P;
P.getrollno (626);
P.getmak (116,282);
P.gettotal ();

P.showrollno ();
P.showmarks ();
P.showtotal ();
}
```

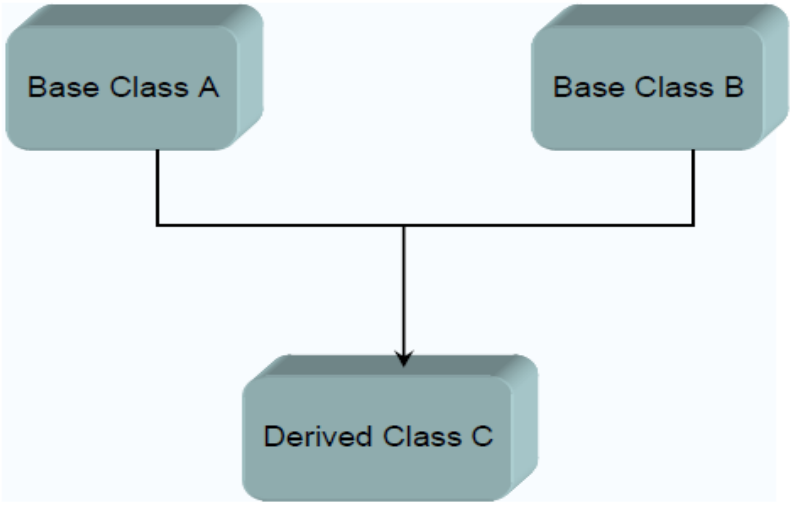
3. MULTIPLE INHERITANCE

Multiple inheritance refers to a feature in which a class can inherit behaviors and features from more than one superclass (base class). This contrasts with single inheritance, where a class may inherit from only one superclass.

Only few Object Oriented Programming Languages supports Multiple Inhritance because of some Ambiguities arise in multiple inheritance.

Languages that mostly support multiple inheritance are: Eiffel, C++, Python, Perl, and CLOS. Java and C# do not allow multiple inheritance; this results in no ambiguity. However, Java and C# allow classes to inherit from multiple interfaces.

The class hierarchy shown below represents Multiple Inheritance.



Syntax:

Class *base_class_name1*
{
// List of members

};
Class *base_class_name2*
{
// List of members
};
Class *derived:* *<visibility mode> base_class_name1, < visibility mode> base_class_name2*
{
// List of members
};

Note:
In programming, when a new class is derived from two or more base classes Then after Colon Symbol all base class names with its respective visibility Modes should be separated by Comma (,) Symbol as represented below.

Class derived: <visibility mode> base_class_name1, < visibility mode > base_class_name2

Example 2:
Consider the same program in example 1

```
#include <iostream.h>
class A
{
protected:
int roll_no;
public:
void getrollno (int r)
{
roll_no=r;
}
void showrollno ()
{
cout <<"student roll number: "<<roll_no<<"\n";
}
};
class B
{
protected:
float math, cs;
public:
void getmarks (float x, float y)
{
math=x;
cs=y;
}
```

```
void showmarks ()
{
cout<<"The marks in mathematics = "<< math<<"\n";
cout<<"The marks in computer science = "<<cs<<"\n";
}
};
class C: public B, public A
{
float total;
public:
void gettotal()
{
total = math + cs;
}
void showtotal()
{
cout<<"total marks="<<total;
}
};
void main ()
{
int f;
float s1, s2;
C P;

cout<<"\nenter the Roll no of Student\n";
cin>>f;
cout<<"\nenter the marks of the student in 2 subject\n";
cin>>s1>>s2;
P.getrollno(f);
P.getmarks(s1,s2);
P.gettotal();
P.showrollno();
P.showmarks();
P.showtotal();
}
```

In the above example program the functions of class A i.e. getrollno (int r) & showrollno () and functions of class B i.e. getmarks (float x, float y) & showmarks () are inherited means these functions are declared in their respective classes but now these functions representing as it belongs to class C, by the statement as mentioned above the compiler represents all the functions of these three functions in only one class i.e.. Class C because in the main function, all the functions are invoked using object of class C, So a compiler reads only class C from which it invokes all the functions.

❖ Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```
#include <iostream.h>
class A
{
public:
void display (void)
{
cout <<"class M\n";
}
};
class N
{
public:
void display (void)
{
cout <<"class N\n";
}
};
```

Which **display ()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```
class P:public M, public N
{
public:
void display (void)
{
M::display ();
}
};
int main ()
{
P p;
p.display ();
}
```

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```
#include <iostream.h>
class A
{
public:
void display (void)
{
cout <<"class A\n";
}
};
class B: public A
{
public:
void display (void)
{
cout <<"class B\n";
}
};
```

In this case, the function in the derived class overrides the inheritance function and therefore, a simple call to **display ()** by **B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

```
int main ()
{
B p;
p.display ();
p.A::display();
p.B::display();
}
```

8.7 Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

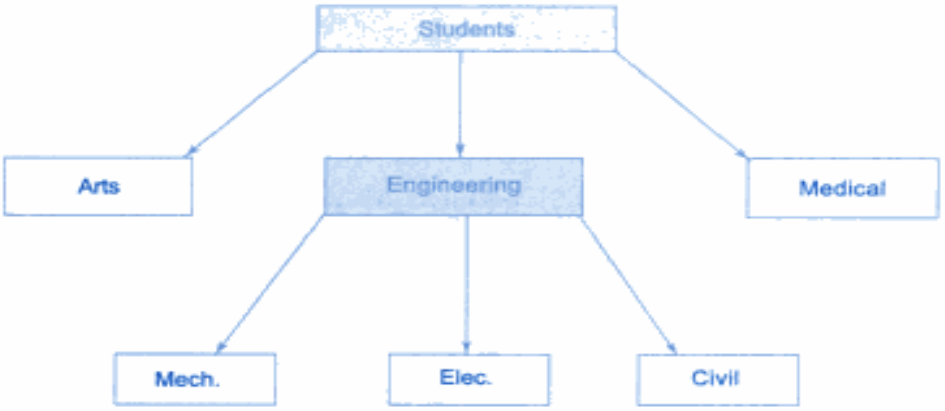


Fig. 8.9 ⇔ *Hierarchical classification of students*

Example: Now we will implement a sample example which demonstrates the hierarchical Inheritance.

```
#include<iostream.h>
class A
{
public:
int x, y, z;
void getdata ()
{
cout<<"enter the values of x, y\n";
cin>>x>>y;
}};
class B: public A
{
public:
void add ()
{
z= x + y;
}
void showadd ()
{
cout<<"the add result: "<<z<<"\n";
}};
```

```
class C: public A
{
public:
void mul()
{z= x*y;}
void showmul()
{
cout<<"The multiply result: "<<z<<"\n";
}
};
void main ()
{
B R;
R.getdata();
R.add();
R.showadd ();
C p;
p.getdata();
p.mul();
p.showmul();
}
```

8.8 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in Sec. 8.5. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called **sports**. The new inheritance relationship between the various classes would be as shown in Fig. 8.11.

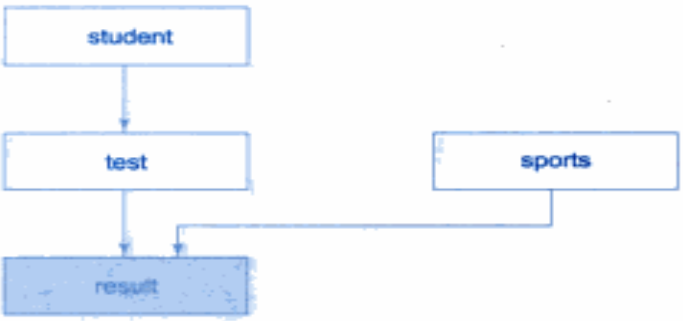


Fig. 8.11 ⇔ *Multilevel, multiple inheritance*

As I said hybrid inheritance is a mix of two or more types of inheritance, in our example now we mixed Multi-Level and Multiple Inheritance. In the above class hierarchy observe the chain of classes student, test and result represents multi-level inheritance and class hierarchy between classes Test, sports and result represents Multiple Inheritance where result class derived from two classes i.e. test and sports classes.

Now the below example program illustrates the implementation of both multiple and multi-level inheritance.

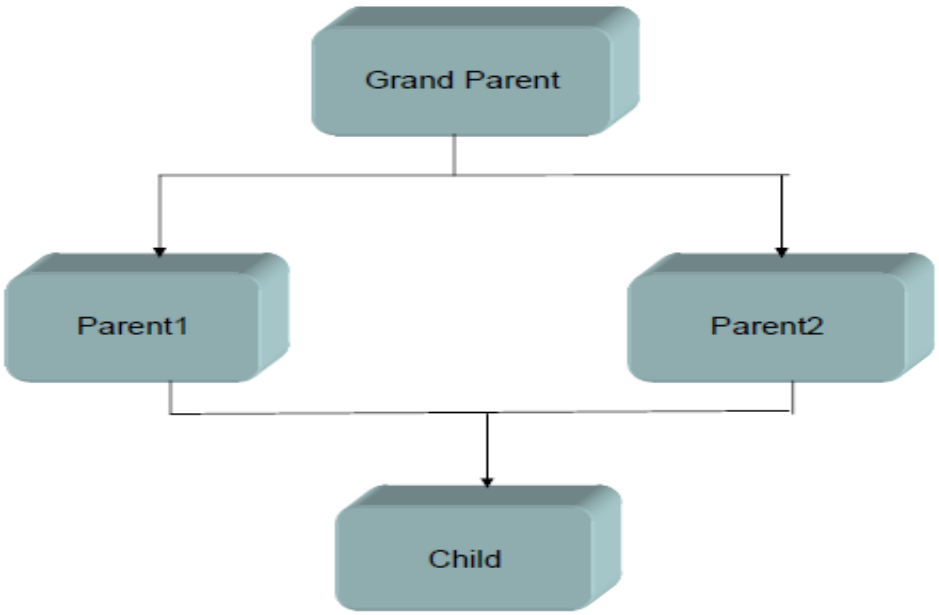
```
#include<iostream.h>
class student
{
protected:
int roll_No;
public:
void getrollno(int r)
{
roll_No=r;
}
void showrollno()
{
cout <<"student roll number: "<<roll_No<<"\n";
}
};
class test: public student
{
protected:
int math, cs;
public:
void getmarks (int x, int y)
{
math=x;
cs =y;
}

void showmarks ()
{
cout<<"The marks in mathematics = "<< math<<"\n";
cout<<"The marks in computer science = "<<cs<<"\n";
}
};
class sports
{
protected:
float score;
public:
void getscore(float s)
{
score=s;
}
void showscore()
{
cout<<"score= "<<score<<"\n";
}
};
class result: public test, public sports
{
float total;
public:
void display (void);
```

```
};
void result:: display()
{
total= math+cs+score;
showrollno();
showmarks();
showscore();
cout<<"total score:"<<total<<"\n";
}
int main ()
{
result P;
P.getrollno(626);
P.getmarks(90,42);
P.getscore(60.4);
P.display();
return 0;
}
```

8.9 Virtual Base Classes

We have just discussed 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 illustrated in Fig. 8.12. The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.



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 A                              // grandparent
{
      .....
      .....
};
class B1 : virtual public A          // parent1
{
      .....
      .....
};
class B2 : public virtual A          // parent2
{
      .....
      .....
};
class C : public B1, public B2       // child
{
      .....                  // only one copy of A
      .....                  // will be inherited
};
```

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 the virtual base class and a derived class.

note

The keywords **virtual** and **public** may be used in either order.

For example, consider again the student results processing system discussed in Sec. 8.8. Assume that the class **sports** derives the **roll_number** from the class **student**. Then, the inheritance relationship will be as shown in Fig. 8.13.

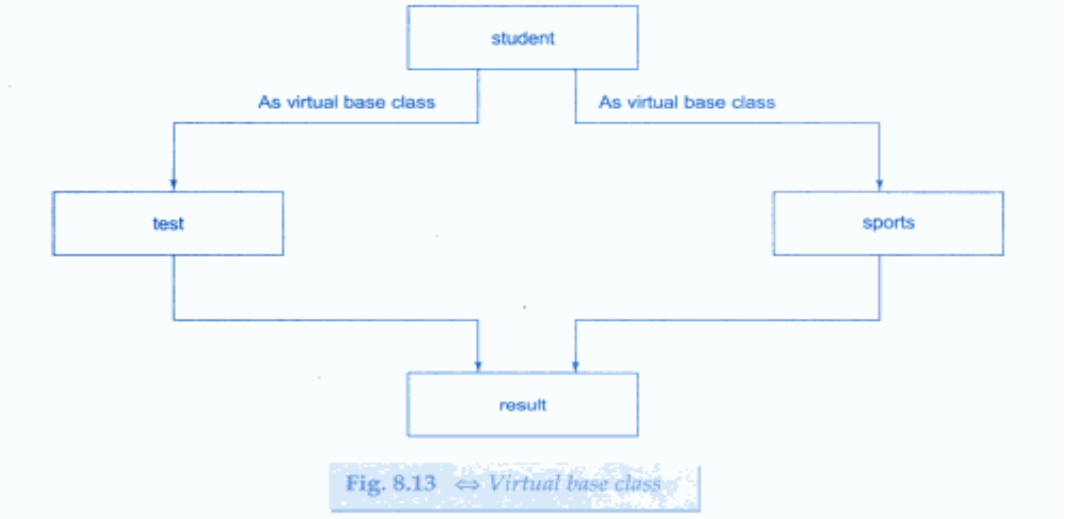


Fig. 8.13 ⇔ Virtual base class

A program to implement the concept of virtual base class is illustrated as bellow:

```
#include<iostream.h>
class student
{
protected:
int roll_No;
public:
void getrollno(int r)
{
roll_No=r;
}
void showrollno()
{
cout <<"student roll number: "<<roll_No<<"\n";
}
};
class test: virtual public student
{
protected:
int math, cs;
public:


void getmarks (int x, int y)
{
math=x;
cs =y;
}

void showmarks ()
{
cout<<"The marks in mathematics = "<< math<<"\n";
cout<<"The marks in computer science = "<<cs<<"\n";
}
};
class sports: public virtual student
{
protected:
float score;
public:
void getscore(float s)
{
score=s;
}
```

```
void showscore()
{
cout<<"score= "<<score<<"\n";
}
};
class result: public test, public sports
{
float total;
public:
void display (void)
{
total= math+cs+score;
showrollno();
showmarks();
showscore();
cout<<"total score:"<<total<<"\n";
}};
void main ()
{
result P;
P.getrollno(626);
P.getmarks(90,42);
P.getscore(60.4);
P.display();
}
```