UNIT-3

** Class :

→ A class is a user defined datatype.

→ A class is a way to bind the data and its associated function together.

→ It is collection of variables called datamembers and functions called as member functions.

→ The data members and member functions are collectively known as members of a class.

→ A class is similar to a structure in C.

→ A class specification has 2 parts.

→ ~~A class~~ (1). class declaration.

(2). class function definition.

→ The class declaration specifies the type and scope of its members.

→ The class function definition describes how the class functions are implemented.

→ The class declaration is similar to a struct declaration.

→ The keyword class is used to create a classname.

→ The body of a class is enclosed with in braces and terminated by a semicolon.

→ Data members & member functions are grouped under two sections namely, Private and Public.

→ The keywords Private and Public are known as visibility labels.

→ Note that these keywords are followed by a colon.

→ which data members & memberfunctions declared in the Private that can be ~~use~~ accessed only from with in the class.

→ Public class members can be accessed from outside the class

→ The use of the keyword Private is optional. By default the members of a class are Private.

- The general form of class declaration is

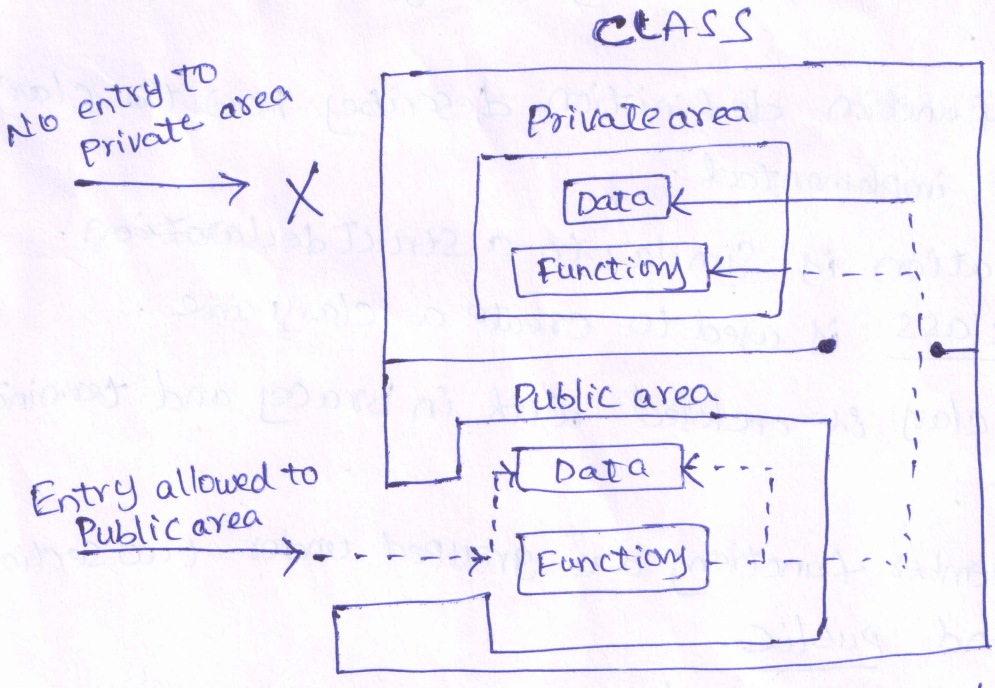
```
class class-name
{
 Private :
        variable declarations;
        function declarations;
 Public :
        variable declarations;
        function declarations;
};
```

Eg:

```
class item
{
 Private :
        int number;
        float cost;
 Public :
        void getdata(int a,
                     float b);
        void putdata();
};
```

** • The keywords private and public are also known as access specifiers.

Data Hiding in classes: -



→ The binding of data and functions together into single class type variable is refered to as Encapsulation.

—

Object:

→ Objects are basic runtime entities.
→ Object is a instance of a class, and also class variable.
→ once a class has been defined, we can create any number of objects using that class name.
→ They may represent a person, a place, any item that the program has to handle.
→ creating an object actually allots memory for storage of data members.
→ The data members and member functions reside at the specific memory location created by an object.

Creating Objects:

→ once a class has been declared, we can create variables of that type using the class name they are called objects.

Syntax: <classname> <objectname>;

eg: student S1;

① class item
{

} x, y, z;
objects of item class.

② item x, y, z;

Accessing class members using Objects:

→ The member functions of a class can be accessed using the objects of that class.

Syntax: 1. objectname . functionname (arguments list);
2. <objectname> . <member-functionname>;

eg: S1 . Getdata();
S1 . Putdata(2,4,3);

- A variable declared in public can be accessed by the objects directly.

Example program to demonstrate the class & objects:

```
#include <iostream>
using namespace std;          } → Header file used in C++

class student ——————→ creation of student class
{
  private :
        int rollno, m1, m2, m3;
        float total, avg;          } → data members.
        char name[20];
  public :
        void getdata()  ——→ member function
        {
        cout << "\n enter student name";
        cin >> name;
        cout << "\n enter Roll number";
        cin >> rollno;
        cout << "\n marks in 3 subjects";
        cin >> m1 >> m2 >> m3;
        }
        void calculate()
        {
          total = m1+m2+m3;
          avg = total/3;
        }
        Void display()
        {
        cout << "Name is" << name;
        cout << "Roll Num is" << rollno;
        cout << "marks in subject1" << m1;
        cout << "marks in subject2" << m2;
        cout << "marks in subject3" << m3;
```

Access specifier

```
    cout<<"total is"<<total;
    cout<<"Average is"<<avg;
  }
};
  int main()
  {
    student s;  → creation of object s
    s.getdata();
    s.calculate();   } → Accessing of member functions.
    s.display();
  }
```

- The above program
```
void getdata();
void calculate();   } → Member functions.
void display();
```

** Defining member functions:

→ Member functions can be defined in 2 places.

(1) • out side the class definition.
(2). Inside the class definition.

(1). out side the class definition:-

- Member functions that are declared inside a class that can be defined separately out side the class.
- Their definitions are very much like the normal functions.
- The general form of a member function definition is:

```
<Return type> <class name> :: function-name(argument list)
{
    - - -  function body
    - - -
}
```

** • By using the symbol :: scope resolution operator we have to define the member functions in out side the class definition.

eg:
```
class ABC
{
private:
        int a;
public:
        void getdata();   // function declaration inside the class
};
void ABC::getdata()       // function definition outside the class declaration.
{
    // body of function
}
```

→ The member label <classname>:: tells the compiler that the function function_name belongs to the class <classname>

eg: Program to addition of 2 numbers.

```
#include<iostream>
using namespace std;
class Add
{
private:
        int a,b,sum;
public:
        void getdata();
        void process();
        void display();
};
void Add::getdata()
{
cout<<"enter the value of a & b";
 cin>>a>>b;
}
void Add::process()
{
    sum = a+b;
}
```

```
void Add :: display()
{
  cout<<" sum is : "<< sum;
}
  int main()
  {
     Add abc;
     ~~cin>>~~ abc. getdata();
               abc. Process();
               abc. display();
  }
```

Output: enter the value of a & b 5 6

Sum is 11

Inside the class definition:

→ Another method of defining a member function is inside the class definition.

→ Normally, only small functions are defined inside the class definition.

Syntax:

```
class classname
{
  Private:
     data members;
  Public:
     memberfunction()
     {
        // body of function.
     }
};
```

Ex:

```
class A
{
Private:
     int a;
Public:
     void getdata()
     {
     // Body of function
     }
};
```

Example Program:

```
#include<iostream>
using namespace std;
  class Person
  {
```

```
Private:
        int age;
        Char name[10];
Public:
        void getdata()
        {
        cout << "enter Person name:";
         cin >> name;
         cout << "enter age:";
         cin>>age;
        }
        void display()
        {
        cout << "Name is : " << name;
        cout << "Age is :" << age;
        }
};
void int main()
        {
          Person P;
          P.getdata();
          P.display();
        }
```

output:

```
enter Person name : SriRan
enter age : 20
Name is : SriRan
Age is : 20
```

** Memory Allocation For Objects:

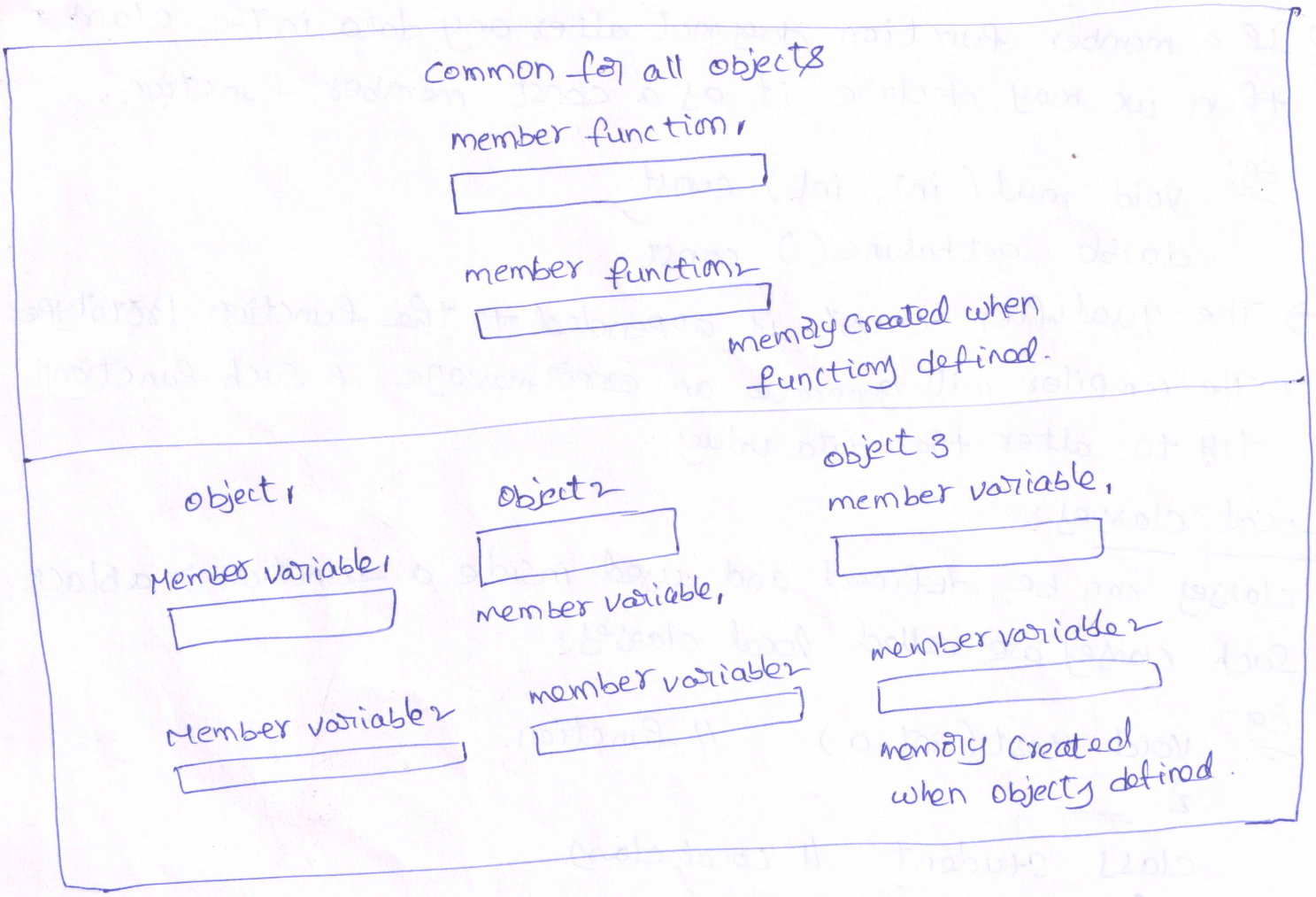
→ Actually, the member functions are created and placed in the memory space only. Once when they are defined as a Part of class Specification.

→ Since all objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.

→ only space for member variables is allocated separately for each object.

→ separate memory locations for the objects are essential because the member variables will hold different data values for different objects.

Object of Memory:



Empty classes:

→ The main reason for using a class is to encapsulate data and code.
→ However, it is possible to have a class that neither has data or code.
→ It is called as an empty class.
→ An empty class is declared as follows.

```
class classname
{
};
```

Ex:

```
class ABC
{
};
```

→ During the initial stages of development of a project, some classes are not fully identified or not fully implemented.
→ In such cases, we have empty classes.
→ Such Empty classes are called "Stubs".

The const member functions:

→ If a member function does not alter any data in the class then we may declare it as a const member function.

Eg: void mul(int, int) const;
double getbalance() const;

→ The qualifier const is appended to the function Prototype.

→ The compiler will generate an error message if such functions try to alter the data values.

Local classes:

→ classes can be defined and used inside a function or a block.

→ Such classes are called local classes.

Eg:
```
void test(int a)    // function
{
  ==
  class student     // local class
  {
    ==
  };
}
```

→ Local classes can use global variables and static variables defined inside the function, but can not use automatic local variables.

** Array of objects:

→ An array is a collection of similar data items stored in continous memory location, represented by single name.
i.e char name[20];

→ Similarly, we can also have arrays of variable that are of the type class.

→ Such variables are called arrays of objects.

→ Since an array of objects behave like any other array,

we can use the usual array accessing method to access individual elements.

→ An array of objects is stored inside the memory in the same way of a multi dimensional array.

→ only the space for data items of the objects is created.

→ Member functions are stored separately and will be used by all the objects.

Eg: Program to demonstrate array of objects.

```
#include<iostream>
using namespace std;
class employee
{
  char name[30];
  float age;
  public:
    void getdata(void);
    void putdata(void);
};
void employee::getdata(void)
{
  cout<<"enter name:";
  cin>>name;
  cout<<"enter age:";
  cin>>age;
}
void employee::putdata(void)
{
  cout<<"Name:"<<name;
  cout<<"Age:"<<age;
}
  const int size=3;
int main()
{ employee manager[size];
  for(int i=0; i<size; i++)
  { cout<<"\n Details of manager"<<i+1<<"\n";
    manager[i].getdata();
```

```
cout<<"\n";
for(int i=0; i<size; i++)
{
cout<<"\n manager"<<i+1<<"\n";
manager[i].putdata();
}
return 0;
}
```

Output:

```
Details of manager 1
enter name : Karthic
enter age : 45
Details of manager 2
enter name : Aswini
enter age : 40
Details of manager 3
enter name : Lavanya
enter age : 35
```

```
Manager 1
Name : Karthic
Age : 45
Manager 2
Name : Aswini
Age : 40
Manager 3
Name : Lavanya
age : 35.
```

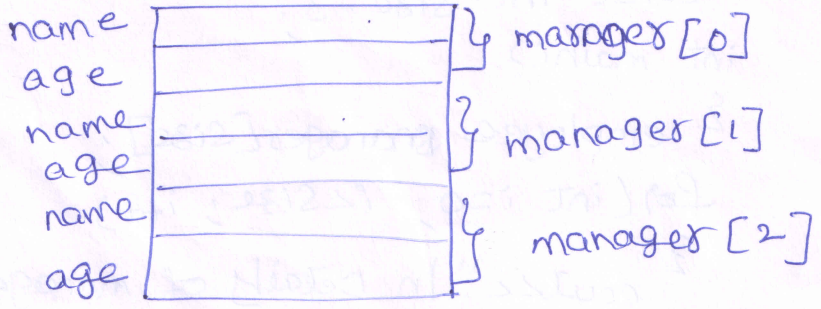
→ The above program:
The identifier employee is a class name. That can be used to create array of object ~~not~~ is manager i.e

```
employee manager[size];
```

→ Accessing of member functions using manager[size] object is follows

```
manager[i].putdata();
manager[i].getdata();
```

→ Storage of data items of an object array is as follows:-



we can use the usual array accessing method to access individual elements.

→ An array of objects is stored inside the memory in the same way as a multi dimensional array.

→ only the space for data items of the objects is created.

→ Member functions are stored separately and will be used by all the objects.

Eg: Program to demonstrate array of objects.

```
#include<iostream>
using namespace std;
class employee
{
  char name[30];
  float age;
  public:
      void getdata(void);
      void putdata(void);
};
void employee::getdata(void)
{
  cout<<"enter name:";
  cin>>name;
  cout<<"enter age:";
  cin>>age;
}
void employee::putdata(void)
{
  cout<<"Name:"<<name;
  cout<<"Age:"<<age;
}
  const int size=3;
int main()
{ employee manager[size];
  for(int i=0; i<size; i++)
  { cout<<"\n Details of manager"<<i+1<<"\n";
    manager[i].getdata();
```

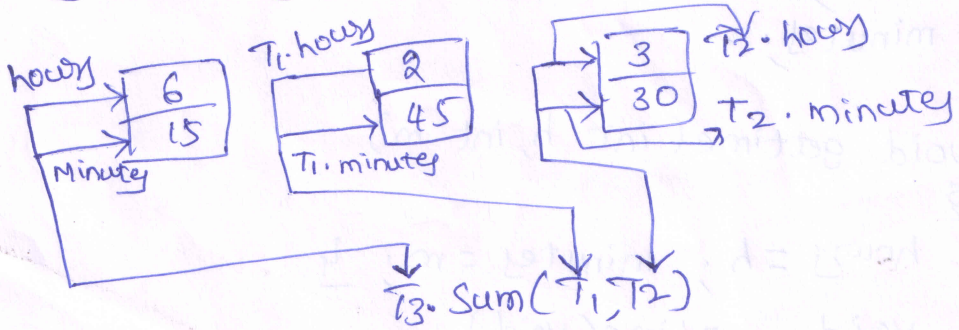
Objects As Function Arguments :-

```
    minutes = t1.minutes + t2.minutes;
    hours = minutes / 60;
    minutes = minutes % 60;
    hours = hours + t1.hours + t2.hours;
}

int main()
{
    time T1, T2, T3;
    T1.gettime(2,45);
    T2.gettime(3,30);
    T3.sum(T1,T2);    // T3 = T1+T2
    cout << "T1 = "; T1.puttime();
    cout << "T2 = "; T2.puttime();
    cout << "T3 = "; T3.puttime();
    return 0;
}
```

output :
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes

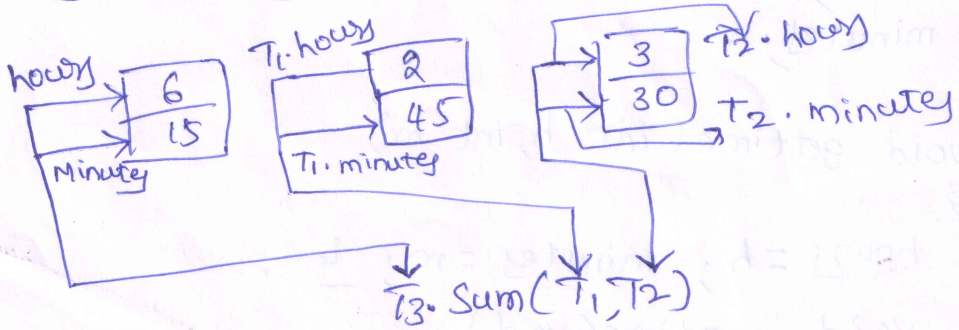
Accessing members of objects with in a called function



```
void sum(time, time);   // declaration of objects as arguments.
};
void time::sum(time t1, time t2)   // t1, t2 are objects.
{
minutes = t1.minutes + t2.minutes;
hours = minutes / 60;
minutes = minutes % 60;
hours = hours + t1.hours + t2.hours;
}
int main()
{
    time T1, T2, T3;
    T1.gettime(2,45);
    T2.gettime(3,30);
    T3.sum(T1,T2);   // T3 = T1+T2
    cout << "T1 = "; T1.puttime();
    cout << "T2 = "; T2.puttime();
    cout << "T3 = "; T3.puttime();
    return 0;
}
```

output: T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes

Accessing members of objects with in a called function



Static Data Members:

→ A data member of a class can be qualified as static

→ The properties of a static member variable are similar to that of a 'c' static variable.

→ A static member variable has certain special characteristics They are

* It is initilized to zero when the first object of its class is created.
* No other initialization is Permitted.
* only one copy of the member is created for the entire class and it is shared by all the objects of that class.
* It is visible only with in the class, but its life time is entire program.
* The static data members are stored seperately, rather than as a part of an object.
* The type and scope of each static member variable must be defined outside the class definition.

Syntax: Static datatype variablename;

Eg: static int count;

Ex:1
```
class item
{
  private:
      static int countt;
  public:
      void getdata();
};
```

→ if we want to assign a value to static variable then syntax is as follows.

int item :: count = 10;

→ Definition of a static data member is

int item :: count

Example Program:

```
#include<iostream>
using namespace std;
class item
{
  static int count;
   int number;
  public:
      void getdata(int a)
      {
        number = a;
        count++;
      }
        void getcount()
        {
        cout<<"count: ";
        cout<< count <<"\n";
      }
};
 int item::count;
int main()
{
item a, b, c;      // count is initilized to zero
a.getcount();      // display count and count value incremented
 b.getcount();     //    ''
 c.getcount();     //    ''
a.getdata(100);    // getting data in to object a
b.getdata(200);    //  ''     ''   ''  object b
 c.getdata(300);   //  ''     ''   ''  object c
cout<<"After reading data"<<"\n";
   a.getcount();   // display count.
   b.getcount();
   c.getcount();
    return 0;
}
```

output:

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

Static Member function: Static Object:

→ Like static member variable, we can also have static member functions.

→ A member function that is declared as static has the following properties.

* A static function can have access only other static members, declared in the same class.
* A static member function can be called using the class name.
* Syntax for declaring static member function.

```
classname::functionname.
```

→ In the below program, the static function showcount() displays the numbers of objects created till that moment.

→ Declaration is as follows

```
static void(datatype) showcount()   // showcount ← function name
{
   // Body of the function     // code is not static
}
```

Ex: Program:

```
#include <iostream>
using namespace std;
class test
{
   int code;
   static int count;          // static member variable
public:
   void setcode()
   {
     code = ++count;
   }
   void showcode()
   {
     cout << "object number : " << code << "\n";
   }
```

```
static void showcount(void)   // static member function
  {
    cout<<"count:"<<count<<"\n";
  }
};
int test::count;   // static variable definition
int main()
{
  test t1,t2;
  t1.setcode();
  t2.setcode();
  test::showcount();   // accessing static function
  test t3;
  t3.setcode();
  test::showcount();
  t1.showcode();
  t2.showcode();
  t3.showcode();
  return 0;
}
```

output :

```
count : 2
count : 3
Object number : 1
object number : 2
object number : 3.
```

(v) Friend Functions :

→ C++ allows the common function to be made friendly with both the classes, there by allowing the function to have access to the private data of these classes.

→ Such a function need not be a member of any of these classes.

→ To make an outside function friendly to a class, we have to simply declare this function declaration, preceeded by the keyword friend.

→ The function is defined else where program like any normal C++ function.

→ The function definition does not use either the keyword "friend" or the scope resolution operator (::)

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

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

→ Syntax:

```
class ABC
{
=
public:
    friend void add();
};
```

Ex:2 Member function of one class is friend function of another class.

→ it is defined by the scope resolution operator.

Ex

```
class x
{
=
int fun();
};
```

```
class y
{
=
friend int x::fun();
};
```

Characteristics of friend functions:

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

→ It cannot be called using the objects of that class.

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

→ It can be declared either in public (or) private section.

→ Usually it has the objects as arguments.

Eg: Program

```
#include<iostream>
using namespace std;
class sample
{
  int a,b;
public:
  void setvalue()
  {
    a=25;
    b=40;
  }
  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)
<<"\n";
return 0;
}
```

output: mean value=32.5

Volatile member function:

→ Like the const member function, we can have the volatile member function.

→ C++ also supports volatile member functions

→ we can declare a member function with the volatile specifier to ensure that, it can be called safely for a volatile objects.

eg:
```
class B
{ int x;
Public: void add() volatile;
};
int main()
{
Volatile B b; // b is a volatile object
b.add(); // call a volatile member function safely.
```

→ The object b is declared volatile.

→ Calling a non-volatile member function from this object is unsafe.

→ To ensure that add() can be called safely for a volatile object, it is also declared volatile.

Overloading Member functions:

→ Functions can be overloaded.

→ This means that multiple member functions can exist with the same name, but with different parameters

→ If a member in the derived class share the same name with members of the base class, they will be hidden to the compiler.

→ constructors and other class member functions, except the destructor can be overloaded.