OBJECTS & CLASSES

Class is a specially designed data type in C++ to hold both the data & function. class declaration is similar to a struct declaration. Class & struct can be used interchangeably with minor modifications. Although use of struct is intended to hold only data.

Class declaration

In a class declaration, the data & its associated functions are mentioned together. Class is an abstract data type. Class declaration consist of

1. Class declaration.
2. Class function definition.

The general form of class declaration is

```
class class_name
{

        private : variable declaration;
                  function declaration;
        public  : variable declaration;
                  function declaration;
};
```

Members : functions & variables are called members. Formally they are referred to as data member & member function respectively.

Visibility label : The keyword private , member belongs to it can be accessed Only from within the class, & public , the members of which are accessible outside the class too.
(Another visibility label protected by all means identical with private only difference being protected members of a class are inheritable, private members are not)

- Default declaration of members in a class is private.
- Members declared in default are hidden outside the class which is Information hiding.
- The binding of data & functions together in to a single class-type variable is referred to as Encapsulation

Note :
1. Only difference between a class & struct in C++ is that by default the members of a class are private but the members of a struct are public.
2. Data hiding is not security technique to protect database , rather it is designed to protect members from accidental manipulation. Private data can be accessed by public functions that operate on data.

Object creation

Once a class has been declared & identified, variables of that type can be Created using class-name (like any other built in type).

```
class_name    obj1, obj2,obj3 …objn;
```

any number of objects can be created now or later with reference to class_name.

Class specification provides only the template & doesn't create any memory space fro object (like struct declaration). Space is allocated during declarations of object.

Objects can also be created using following declaration:

```
Class class_name
{
        ………. ;
        ………. ;
} list of objects;
```

Class function definition

Member functions can be defined

1. Outside the class definition
2. Inside the class definition.

Member function definition are very much like normal functions. They should have a function header and a function body.
The actual code for the function may be contained within the class specifications. However it is possible to declare a member function & define it elsewhere.

An important difference between a member function & normal function is that a member function incorporates member-ship identity label in the header. This label tells the compiler in which class the function belongs to.

Because different class can have member functions with the same name, we must specify the class name when defining a member function.

{ Note : since C++ is strict with regard to type compatibility, two functions with same name (identifier) are distinguished through type of function arguments (Function overloading) }

When the function code is contained within class specification member ship label is specified immediately.

However when it is declared outside the class the general form:

```
return type   class_name : : function_name (argument declaration)
{
            function body;
}
```

the membership label class : : name tells the compiler that the function name belongs to the class class_name. That is the scope of the function is restricted to the clas_name specified in the header line. The symbol :: is called scope resolution operator.
e.g.

```
class date
{
      int   d, m, y;
      void init(int  dd, int  mm, int  yy );
      void add_year(int  n);
      void add_month(int n);
      void add_date(int n);
};

void date:: init(int  dd, int  mm, int  yy)
{
      d=dd;
      m=mm;
      y=yy;
}
```

Member functions defined inside a class are created as inline functions by default. Functions defined outside the class are not normally inline.

[An **inline function** is function that is expanded inline when it is invoked. That is compiler replaces the function call with the corresponding Function code. The inline functions are defined as:

```
inline function  header
{
      function body;
}
```

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 in to stacks & returning to the calling Function. When a function is small, a substantial percentage of executing Time may be reduced by defining them inline.

However we should exercise restraint before making a function inline. The speed benefits of inline functions diminish as the function grows in size. At some point overhead of the function becomes small compared to the function execution and benefits of inline functions are lost.
Inline keyword merely sends a request & not Command to the compiler which may ignore it if function definition is too Large.]

Thus special characteristics of member functions are :

i) several different classes can use the same function name. The membership identity label will resolve their scope.
ii) Member functions can only access the data of the class. (an exception in this rule is a friend function)
iii) A member function can call another member function.

ACCESS CONTROL

Usually data within a class is private & the functions are public so that data is hidden & protected to accidental manipulation & functions that operate on data are accessible from outside the class. However this is not rule & in some circumstances private functions & public data are meaningful. A struct is simply a class whose members are public by default.

e.g.

```
inline void date:: add-year( int  n)
{
        y+=n;
}                         //       is valid.
```

Non member functions are barred from using private members e.g.

```
Void    timewarp ( date  &d )
{
        d.y -=200;    //    error   date   ::y   is  private
}
```

CALLING MEMBERS:

Following is the format to access member functions :

```
Object_name . function_name(actual arguments);
```

(Private data of a class can be accessed only through the member functions & public data as object _name . variable _name)

Member function is called to invoke a specific object of the class, not on the class in general i.e. member functions of a class can be accessed only by an object of the class.

To use a member function, dot operator connects the object name & member function. The dot operator is called <u>class member access operator</u> .

STATIC DATA MEMBER

A data member of a class, which is qualified as static, will have following Features :

i > It is initialized to zero when the first object of its class is created. No other initialization is permitted during object creation(default). However assignment of any value to members during creation is possible .

ii > Only one copy of that member is created for the entire class & is shared by all the objects of that class, irrespective of number of objects.

iii> It is visible only within the class, but its lifetime is the entire Program.

They are normally used to maintain rules common to the class e.g. occurrence of objects. The type &scope of each static member variable must be defined outside the class definition. This is necessary for static data members are stored separately rather than the part of the object.
Static variables are like non-inline member function in that they are declared within class & defined elsewhere. Although default initialization of static member is value ‘0’, assignment of value during defining them is permissible.

```
int   item : : count= 10;
```

STATIC MEMBER FUNCTION

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

i> A static function can have access to only other static members (Functions or variables) declared in the same class.

ii>	A static member function can be called using the class-name (instead of its object) as follows :

class name : :function name

Example program :

```
#include< iostream.h >

class   test
{
          int     item;
          static int count;
     public :
          void setcode(void)
          {
                    item=++count;
          }
          void showcode(void)
          {
                    cout<<" Object No :"<<item<<endl;
          }
          static void  showcount(void)
          {
                    cout<<"count:="<<count<<endl;
          }
};

int   test : :count;

int main(void)
{
          test     t1, t2;
          t1.setcodde();
          t2.setcode();
          test : : showcount();
          test   t3;
          test :: showcount();
          t1.showcode();
          t2.showcode();
          t3.showcode();
          return 0;
}
```

OBJECT AS FUNCTION ARGUMENT

Like other data type objects can also be passed as function arguments. The following procedures are allowed :

1. Pass by value : a copy of the entire object is passed to that function.

2. Pass by reference : only the address of the object is transferred to function.

In the first method (called pass-by-value)the changes made to the object inside the function won't effect the object used to call the function. However that is affected in the second case.

Example program :

```
#include< iostream.h>

class    time
{
         int   hour, minute;
     public :
         void gettime ( int  h, int  m)
         {
                  hour=h;
                  minute=m;
         }
         void puttime(void)
         {
                  cout<< hour<<"hours   and"<<minute<<"minutes";
                           cout<<endl;
         }
         void sum( time, time);
};

void    time:: sum(time   t1, time t2)
{
         minute= t1.minute+ t2.minute;
         hour=minute/60;
         minute=minute%60;
         hour=hour + t1.hour + t2.hour;
}
```

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

Re discussion of the above concept using operator overloading

```
                time    operator + (time);     // prototype

                time   time : : operator +(time t)
                {
                        time  temp;
                        temp . minute= minute+ t . minute;
                        temp . hour  = temp . minute/60;
                        temp . minute = temp . minute % 60;
                        temp. hour=temp .hour + hour + t.hour;
                        return(temp);
                }
```

once the operator is overloaded the statement   t1=t2+t3 can be invoked.

ARRAY OF OBJECTS

Like array of structured variable, array of objects concept is crucial to maintain multiple objects at run-time. The striking features of array remain intact for objects as well i.e. partitioned area for various objects will be allocated in successive memory allocations and each element of the array will be referenced w. r. t an indexed set . However importance must be given before using array of objects to memory space-requirement. High space is required for Objects in comparison with other data type for objects truly represent the real-life entity and hence all the information are incorporated within it.

Array of objects can be declared as :

```
                class _name      array _name[size];
```

size is a constant integral parameter and any finite number of objects can be declared.
Any array object can be accessed using counter & any member can be invoked using class membership operator (.) similar other object.

```
         array _name [counter _value] . data _member;
  array _name [counter _value] . member function( argument list);
```

```
        /*   accessing objects used as array elements */

#include< iostream.h>

class  student
{
        char *name;
        int  roll;
    public :
            void get(void)
            {
                 cout<<endl<<" Enter name :";
                 name=new char[20];
                 cin>>name;
                 cout<<"\nEnter roll :";
                 cin>>roll;
            }
           void  show(void)
           {
                 cout<<endl<<name<<"\t"<<roll;
           }
};
int  main(void)
{
        student    stud[20];
        int   n,i;
        cout<<"\n How many students :";
        cin>> n;
        for ( i=0; i<n ; i++)
                stud[i].get();
        for ( i=0; i<n ; i++)
                stud[i].show( );
        getch();
        return 0;
}
```

FRIEND FUNCTION

To make an outside function ‘friendly to a class, we have to simply declare this function as a friend of the class as :

```
        class  ABC
        {
                        ……….;
                public :
                        ……….;
                        friend void fn(void);
        };
```

The function supposed to be normal, is defined elsewhere in the program.

Sample program

```
#include<iostream.h>

class sample
{
                int   a;
                int   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(void)
{
        sample      X;
        X . setvalue();
        cout<< “ \nMean Value :” << mean(X)<<endl;
        getch( );
        return 0;
}
```

Member functions of one class can be friend to another class. On that they are defined using scope resolution operator.

```
Class X                                         class  Y
{                                               {
          ……….. ;                                         ………… ;
          ……….. ;
                                                          …………. ;
          int   func1( );                                 friend X::func1( );
          ……….. ;                               };
};
```

We can also declare all the member functions of a class is friend to another class as

```
                              class  Z
                              {
                                        ……………. ;
                                        friend  class X;
                              };
```

Sample program

```
#include<conio.h>
#include<string.h>
#include<iostream.h>
#include<stdio.h>

const int size=20;

class XYZ;

class ABC
{
          char name[size];
          int  age;
          public : void get(void)
                    {
                              cout<<endl<<" Name  :";
                              fflush(stdin);
                              cin>>name;
                              cout<<" Age :";
                              cin>>age;
                    }
                    void show(void)
                    {
                              cout<<name<<"\t"<<age<<endl;
                    }
                    int max(XYZ);
};
```

```
class XYZ
{
        char name[size];
        int  age;
        public : void get(void)
                        {
                                cout<<endl<<" Name :";
                                fflush(stdin);
                                cin>>name;
                                cout<<" Age :";
                                cin>>age;
                        }
                        void show(void)
                        {
                               cout<<name<<"\t"<<age<<endl;
                        }
                        friend int ABC::max(XYZ);
};

int ABC::max(XYZ p)
{
        if(strcmp (name, p.name)<0)
                        return(0);
        else
                        return(1);
}

int main(void)
{
        ABC    ab;
        XYZ    xy;
        int     p;
        ab. get();
        xy. get();
        p=ab. max( xy );
        if(p==0)
                ab. show( );        // to print the object whose name alphabetically
        else                        // preceeds
                xy. show( );
        getch( );
        return 0;
}
```

A friend function possesses certain special characteristics as follows:

- It is not in the scope of the class to which it is has been declared as friend.
- Since it is not on the scope of the class, it can't be called using object of that class. It can be invoked like a normal function.
- Unlike member function, it can't access the member name directly and has to use an object name & dot membership operator with each member name.
- It can be declared either in public or in the private part of a class without affecting its meaning.
- Usually it has objects as argument.

A friend function can be called by reference. In this case the local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.
Use of friend function is necessary in operator overloading too.

```
        /* program emphasizing passing of multiple class objects
                          & return-by refernce */

#include<iostream.h>
#include<stdio.h>
#include<conio.h>
const int size=20;

class ABC;

class XYZ
{
        char name[size];
        int  age;
        public : void get(void)
                        {
                                cout<<endl<<" Name :";
                                fflush(stdin);
                                cin>>name;
                                cout<<" Age :";
                                cin>>age;
                        }
                        void show(void)
                        {
                               cout<<name<<"\t"<<age<<endl;
                        }
                        friend   int  &max (XYZ &, ABC &);
```

```
                              int   get_age(void)
                              {
                                      return(age);
                              }
};

class ABC
{
          char name[size];
          int  age;
          public : void get(void)
                              {
                                       cout<<endl<<" Name  :";
                                       fflush(stdin);
                                       cin>>name;
                                       cout<<" Age :";
                                       cin>>age;
                              }
                              void show(void)
                              {
                                      cout<<name<<"\t"<<age<<endl;
                              }
                              friend int & max(XYZ &,ABC &);
};
int  & max( XYZ  &m, ABC &n)
{
          if(m.age>n.age)
                    return(m.age);
          else
                    return(n.age);
}

int main(void)
{
           ABC  ab;
           XYZ xy;
           clrscr();
           ab.get();
           xy.get();
           if(max(xy,ab)==xy. get_age( ))         // to print the object
                            xy. show();                   // having higher age
           else
                            ab. show();
           getch();
           return 0;
}
```

CONST MEMBER FUNCTION

If a member function doesn't alter any data in the class then we may declare it as a constant member function as follows:

```
                    void     mult( int, int)    const;
                    double get_balance( )  const;
```

The qualifier const is appended to the function prototypes (in both declaration & definition). Compiler will generate an error message if such functions try to alter the data values.

MEMBER DEREFERENCING OPERATORS

: :*	poiter to member declarator
.*	pointer to member access operator
->*	member dereferencing operator.

POINTERS

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name.

A class member pointer can be declared using the operator : :* with the class name. E.g.

```
          class   A
          {
                    private :   int   m;
                    public :
                              void show( );
          };
```

We can define a pointer to the member m as follows :

```
                              int    A : :* ip = & A ::m;
```

the ip pointer created thus acts like a class member in that it must be invoked with a class object, the phrase A::* means "pointer to member of a class". The phrase & A : : m means " address of the m member of a class". Remember int *ip=&m won't work . this is because m is not simply integer data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer ip can now be used to access the member m inside members (or friend functions). Let us assume that a is an object of A declared in a member function. We can access m using the pointer ip as follows:

```
                cout<<  a.*ip;
                cout<< a.m;
```

Now consider

```
                A    *ap;
                ap = &a;
                cout<< ap->*ip;
                cout<<ap->m;
```

The dereferencing operator ->* is used to access a member when we use pointers to both the object & the member.

The dereferencing operator .* is used when the object itself is used with the member pointer.
Note that *ip is used like a member name.

We can also design pointers to member functions which then can be invoked using the de referencing operators in the main as

(object_ name.*pointer to member function)(argument);

(object_ pointer ->*pointer –to-member)(argument);

the precedence of ( ) is higher than that of .* and ->*, so the parenthesis are necessary.

```
#include< iostream . h>

class M
{
                 int   x, y;
        public  :  void setval ( int  a, int  b)
                {
                      x=a;
                      y=b;
                }
                friend int sum(M);
};
```

```
int  sum(M  m)
{
        int   M : : *px = & M : :x ;
        int   M : : *py = & M : :y ;
        M *pm = & m;                    //    object-pointer
        int  s = m.*px + px->*py;     //  member ptr  through object name
        return(s);                      //  & object pointer
}

int   main(void)
{
        M  n;
        void   ( M : : * pf) (int , int ) = & M : : setval ;
        (n.* pf)(10,20);
        cout<< " Sum="<< sum(n)<< endl ;
        M   *op= & n ;
        (op->*pf ) (30 ,40 );
        cout<< "sum="<<sum(n)<<endl;
        return(0);
}
```

SAMPLE PROGRAM (on pointer to object)

```
#include< iostream . h>

class Distance
{
        private  :  int      feet ;
                    float   inches ;
        public  :  void  getdist ( void)
                   {
                        cout<< "\n Enter   feet  :";
                        cin >> feet;
                        cout<< "\n Enter   inches :";
                        cin >> feet;
                   }
                  void  showdist ( )
                  {
                        cout<< feet <<"\'-"<<inches<<'\" ';
                  }
} ;
```

```
void main( )
{
        Distance  *disptr ;
        disptr  =  new  Distance ;
        disptr -> getdist ( );
        disptr -> showdist ( );
}
```

SAMPLE PROGRAM (on array of pointers to object)

```
#include < iostream . h>
#include<conio.h>

const   int  size= 20;

class  Person
{
        protected :  char    name[size] ;
        public       :  void  get_name(void);
                           void show_name(void) ;
};
void   Person : : get_name(void)
{
        cout<<"\n Enter name :";
        fflush(stdin);
        cin >> name;
}
void Person : : show_name(void)
{
        cout<< endl<<name;
}

int    main ( void )
{
        Person  *pr[size];
        cout<<"\n how many persons :" ;
        for ( int  i=0 ; i < n ; i++)
        {
                pr[i] = new Person;
                pr->get_name( );
        }
        for ( int  i=0 ; i < n ; i++)
                pr[i]->show_name( );
        return 0;
}
```