UNIT – II

Page No. | Date

Classes And objects

WAP to enter student details and display . (outside defi).

```
#include <iostream.h>
#include <conio.h>
class BCA
    { public :-
      int age ;
      char name [20];
   void getdata (void);
   void display (void);
    };

void BCA :: getdata (void)
    {
      cout << "enter :" << endl ;
      cin >> name ;
      cout << " enter age :" << endl;
      cin >> age;
     }
void BCA :: display (void)
    {
      cout << "name :" << name ;
      cout << "age :" << age;
   }
   void main ()
   {
     clrscr();
     BCA a1 , a2,
     a1. getdata ();
```

```
    a2.getdata();
    a1.display();
    a2.display();

    getch();
}
```

WAP to enter student detail and display (inside def).

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

class BCA
    { public:
        int age;
        char name[20];

      void getdata(void)
       {
        cout << "Name: ";
        cin >> name;
        cout << "Age: ";
        cin >> age;
       }
      void display(void)
       {
        cout << "Name: " << name;
        cout << "Age " << age;
       } a1;
```

```
void main()
{
  clrscr();
  a1.getdata();
  a1.display();
  getch();
}
```

WAP : { emp name, sal, add sal, display.
(outside class defination)
5 employer.

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

class employee
   {  char name[20];
      float salary;

      void entry(void);   There will not be garbage value.
      void display(void);
      float expense(int)
   } e[5];

void employee::entry(void)
   {
      cout << "Name: ";
      gets(name);
      cout << "salary: ";
      cin >> salary;
   }
```

```
void Employee :: display(void)
{
    cout << "Name : " << name;
    cout << "\n salary : " << salary;
}
float Employee :: expense(int *e)
{
    int i; float total = 0.0;
    for(i=0 ; i<s ; i++)
    {
        total += e[i].salary;
    }
    return total;
}
void main()
{ clrscr();
  for(int i=0; i<s ; ++i)
  {
      e[i].enter();
  }
  for(int j=0; j<s; ++j)
  {
      e[j].display();
  }
  float sum = e[i].expense(e);
  cout << "Total expense borne : \n";
  cout << sum;
  getch();
}
```

Classname :: func name.

STATIC DATA MEMBER ....

There are certain specific characteristics of a static member variable.

(i) It is initialized to '0' when the first object of its class is created ...

(ii) Only one copy of that member is created for the entire class ..... and is shared by all the objects of that class.

(iii) It is visible only within the class but its lifetime is the entire program.

* Static variables are normally used to ~~hold~~ maintain values to the entire class.

Eg: a static data member can be used as a counter that records the occurances of all the objects

Static member function

* A static member function can be declared only if it fulfills certain properties

(•) A static function can have access to only other static members (variables and function) declared in the same class.

(•) A static member function can be called/ invoked using a class name instead of object.

class name :: func. name;

↓

scope resolution operator

Page No.

by default 0.

The static function is used to keep a count of numbers of object created and maintained by static variable.

Points to keep in mind

1. Static variable ✓
2. Static function ✓
3. Calling of static variable for linking ✓
4. Invoking of static function

Page No.
Date

Program of static variable and function

```
# include < conio.h>
# include < stdio.h>
 class bca
 {       public:
         int x;
         static int count;

         void getdata(void);
         static void display (void);
 };

void bca :: getdata (void)
 {
     cout << " Enter x :";
     cin >> x;
     ++x;
}

void bca :: display (void)
{
     :
     :
     :
 }
}
```

Page No.
Date

Using static variable & member function print 1 to 10 counting.

#
#

```
Class count.
{   public:
    int x;
    static int counting;
    static void print();
};
int count :: counting;
void count :: print()
{       counting ++;
        cout << counting << endl;
}
void main ()
{ clrscr()
    for(i=0; i<10; i++)
    { print(); }
  getch(); Count :: print();
}
}
```

Page No.
Date

THIS POINTER

```
#include <iostream.h>
#include <conio.h>
class bca
    { public:
      void getdata(void)
      {
          cout << "Add :" << this;
      }};
  void main()
  {
    clrscr();
     bca a1, a2, a3;
     a1.getdata();
     a2.getdata();
     a3.getdata();
    getch();
  }
```

This is a unique keyword in c++ which is used to store the address of an object associated with a class.

It represents an object that invokes a member function. 'This' is a pointer that points to object for which 'this' function was called

above program.

CONSTRUCTOR

* It is a special member function whose task is to initilize the object of its class.
* It is special because it has same name as of class name. and it is invoked automatically when the object of its class is created.
* It has some special characteristics :

(a) They should be declared in public section.
(b) They are invoked automatically when the objects are created.
(c) They do not have return types, not even void because they can-not return values.
(d) It can't be inherited but it can be used through a derived class
(e) constructor can-not be virtual.
(f) An object with a constructor or destructor can not be used with a . operator.

Default constructor

* It does not have parameters or if it has parameters, all the parameters have default value.

A constructor is used to state the behaviour of an object and must not have a return type.

Page No.
Date

Parameterized Constructor

* It is used for initializing the argument or parameters
* It is necessary to initize various data elements of different objects with different values when they are created

This is achieved by using and passing arguments to constructor function when the objects are created, so the constructor that can take arguments are called parameterized constructor.

COPY Constructor

It is a constructor which creates an object by initializing it with an object of the same class which has been created previously.

The copy constructor is used to

(a) Initializing one object from another of the same type
(b) Copy an object to pass it as an argument to a function
(c) Copy an object to return it from a function

A copy constructor is a member function which initializes the object using another object of the same class.

Page No.
Date

Syntax

(1) Default constructor

```
Class xyz
{
  public :
  xyz (void)          // constructor declared
{
};                    // constructor defined
void main()
{
  xyz z1;             // constructor invoked
  getch();                automatically when
}                         object created.
```

(2) Parameterized constructor

```
class xyz
{
  public :
  xyz (int, int)      // constructor with
{                         Parameter declared.

};                    // defined.
void main()
{
  clrscr();
  xyz q1 (1,2);       // invoked and value passed
  getch();
}
```

Page No.
Date

3. Multiple constructor

```
class xyz
{
  public:
  xyz(void)
{
}
 xyz(int,char)
{ }
};
void main()
{
  clrscr();
 xyz x1;
 xyz x1(1,a);
 getch();
}
```

Page No.
Date

Program

Default Constructor

```
class employee
{
  public:

 employee (void)
{  float salary;
   cin >> salary;
   cout << salary;
}
};
void main()
{
   elemployee e1;
   getch();
}
```

Parameterized Constructor

```
class employee
{
  public:
employee (int l, int b)
{
  int area;
  area = l*b;
  cout << area;
}
};
```

Page No.

Date

```
void main()
{
   clrscr();
   int c,d;
   cin >> c;
   cin >> d;
   employee e1(c,d);
   getch();
}
```

Page No.
Date

DESTRUCTOR

It is a special member function which destructs or deletes an object. It works just opposite to constructor unlike constructors that are used for initializing an object, destructors destroy the object; similar to constructor the destructor name should exactly match with the class name but it is followed by a tilde sign. (~)

Explicit Call to destructor is only necessary when object is placed at a particular location in memory by using new operator.

When is destructor called ??

1. A destructor function is called automatically when the object goes out of scope.
2. The function ends.
3. The program ends.
4. A block containing local variable ends.
5. A delete operator is called.

* Destructors don't take any argument and don't return anything.

Page No.
Date

Syntax :

```
class xyz
{  public:
   xyz(void);
  ~xyz();
};
```

Program : Destructor (Implicit program)

```
class xyz
{
  public:
  xyz(void)
  { int a;
    cout << "Enter the number";
    cin >> a;
    a++;
    cout << a;
  }
void main()
{
   clrscr();
   xyz x1;
   xyz();
}
```

```
~xyz(void)
{
  cout << "memory freed";
}};
```

Output

Page No.	
Date	

IMPLICIT CALLING

```
#
#
class dest
{ public:
int d, b, area;
 dest ();      { cout << " constructor called ";
 ~ dest ();
void area (void)
{
  cout<< " Enter length";
  cin >> d;
   cout << " Enter breadth";
   cin >> b;
   area = d * b;
   cout << " area is " << area;
 }
  void main()
  {
    clrscr();
    dest x1;          // Constructor invoked.
    x1. area();       // area function
                         invoked.
    getch();
  }
```

FRIEND FUNCTION

Friend function declaration is preceded by the keyword friend. A friend function possesses certain characteristics :

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

2. It is not in the scope of class so it can-not be called using object of that class.

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

```
Eg: friend mean(char);
    :
    void main
    {
        mean(x);
```

4. It can-not access the member names variable. directly and has to use an object name with (.) operator with each member

5. It can be declared either in the public or the private part of class.

6. Friend functions are used in operator overloading

7. Member function of one class can be friend function of another class and in such case they are defined using scope resolution ~~numbe~~ operator.

* All the member function of one class can also be declared as friend function of another class. and that concept is known as friend class.

Merits: friend function never brake security.

→ It can be declared either in the public or the private part of a class.

→ It can be used to increase versatility.

→ It should not be defined in the name of class.

Demerits: It can not showcase runtime polymorphism.

(•) They require extra line of code when we want dynamic binding.

(•) friend function have access to private members

Page No.
Date

FRIEND FUNCTION (explicit calling of destructor)

```
#
#
class abc
{ public :
    int p, q;
  abc();
 ~abc();
  friend void bca ( abc a);      // friend function
};
 void abc :: bca ( abc a)
{
    abc();
    a.~abc();                    // explicit calling
    a.p;
    a.q;
}
int main ()
{
    clrscr();
    abc a;
    bca (a);
    getch();
}
```

Page No.
Date

MULTIPLE CLASS

```
# include <iostream.h>
# include <conio.h>
class A
{
  public:
  int x, y;
  void enter (void)
  {
  cout << "Enter value of x & y :" << endl;
  cin >> x >> y;
  cout << "sum of x & y =" << x+y << endl;
  }
};
  class B
  {
  public:
  int p, q;
  void enter (int, int)
  {
  cout << "Enter value of p & q" << end;
  cin >> p >> q;
  cout << "sum of p & q = " << p+q << endl;
  }
};
  void main ()
{ clrscr();
  int o, m;
  A a1;
  a1. enter()
  B b1;
  b1. enter(o, m)
  getch();
  }
```

FRIEND FUNCTION

```
#include <iostream.h>
#include <conio.h>
class Program
{
  private:
  int a;
  int b;
 public:
void value(void)
{
  a=25;
  b=30;
}
 friend float average(program p)
{
    p.value();
    return(p.a+p.b)/2.0;
}
};
 int main()
 {
   clrscr();
   program x
  x.value();          // Invoke.
  average(x);         // invoke friend function
cout << "The value of average is" << average(x);
  getch();
  return 0;
}
```

UNIT -2

Introduction of Class:

An object oriented programming approach is a collection of objects and each object consists of corresponding data structures and procedures. The program is reusable and more maintainable.
The important aspect in oop is a **class** which has similar syntax that of structure.

class: It is a collection of data and member functions that manipulate data. The data components of class are called data members and functions that manipulate the data are called member functions.

It can also called as blue print or prototype that defines the variables and functions common to all objects of certain kind. It is also known as user defined data type or ADT(abstract data type) A class is declared by the keyword **class**.

Syntax:-

```
class class_name
{

Access specifier :
        Variable declarations;

Access specifier :
        function declarations;


};
```

Access Control:

Access specifier or access modifiers are the labels that specify type of access given to members of a class. These are used for data hiding. These are also called as visibility modes. There are three types of access specifiers

1.private
2.public
3.protected

1.Private:

If the data members are declared as private access then they cannot be accessed from other functions outside the class. It can only be accessed by the functions declared within the class. It is declared by the key word „**private**" .

2.public:

If the data members are declared public access then they can be accessed from other functions out side the class. It is declared by the key word „**public**" .

3.protected: The access level of protected declaration lies between public and private. This access specifier is used at the time of inheritance
Note:-

If no access specifier is specified then it is treated by default as private
The default access specifier of structure is public where as that of a class is “private”

Example:

```
class student
{
private : int roll;
        char name[30];
public:
        void get_data()
        {
                cout<<”Enter roll number and name”:
                cin>>roll>>name;
        }
        void put_data()
        {
                cout<<”Roll number:”<<roll<<endl;
                cout<<”Name        :”<<name<<endl;
        }
};
```

Object:-Instance of a class is called object.

Syntax:

class_name object_name;

Ex:

student s;

Accessing members:-dot operator is used to access members of class

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

Ex:

```
s.get_data();
  s.put_data();
```

Note:

1. If the access specifier is not specified in the class the default access specifier is private
2. All member functions are to be declared as public if not they are not accessible outside the class.

Object:

Instance of a class is called as object.

Syntax:

Class_name object name;

Example:

student s;

in the above example s is the object. It is a real time entity that can be used

Write a program to read data of a student

```
#include<iostream>
using namespace std;
class student
{
private:
        int roll;
        char name[20];


public:
        void getdata()
```

```
{cout<<"Enter Roll number:";
        cin>>roll;
        cout<<"Enter Name:";
        cin>>name;
}
void putdata()
        {cout<<"Roll no:"<<roll<<endl;
          cout<<Name:"<<name<<endl;
}
};
int main()
{
student s;
        s.getdata();
        s.putdata();
        return 0;
}
```

Scope Resolution operator:

Scope:-Visibility or availability of a variable in a program is called as scope. There are two types of scope. i)Local scope ii)Global scope

Local scope: visibility of a variable is local to the function in which it is declared.

Global scope: visibility of a variable to all functions of a program

Scope resolution operator in **"::"** .

This is used to access global variables if same variables are declared as local and global

```
#include<iostream.h>
int a=5;
void main()
{
int a=1;
        cout<<"Local a="<<a<<endl;
        cout<<"Global a="<<::a<<endl;
}
```

Class Scope:

Scope resolution operator(::) is used to define a function outside a class.

```
#include  <iostream>
using namespace std;
class sample
 {
public:
  void output(); //function declaration
};
 // function definition outside the
 class void sample::output() {
  cout << "Function defined outside the class.\n";

};
 int  main() {
 sample   obj;
 obj.output();
 return 0;
}
```

Output of program:
Function defined outside the class.

Write a program to find area of rectangle

```
#include<iostream.h>
class rectangle
{
int L,B;
public:
        void get_data();
        void area();
};

void rectangle::get_data()
{
        cout<<"Enter Length of rectangle";
        cin>>L;
        cout<<"Enter breadth of rectangle";
        cin>>B;
}
int rectangle::area()
{
        return L*B;
}
int main()
{
rectangle r;
        r.get_data();
        cout<<"Area of rectangle is"<<r.area();
return 0;
}
```

INLINE FUNCTIONS:

Definition:

An inline function is a function that is expanded in line when it is invoked. Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. It is defined by using key word **"inline"**

Necessity of Inline Function:

- One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.
- Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.
- When a function is small, a substantial percentage of execution time may be spent in such overheads. One solution to this problem is to use macro definitions, known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.
- C++ has different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function.

General Form:

```
inline function-header
{
        function body;
```

}

Eg:

```
 #include<iostream.h>
inline float mul(float x, float y)
{
        return (x*y);
}
inline double div(double p, double q)
{
        return (p/q);
}
int main()
{
float a=12.345;
float b=9.82;
        cout<<mul(a,b);
        cout<<div(a,b);
        return 0;
}
```

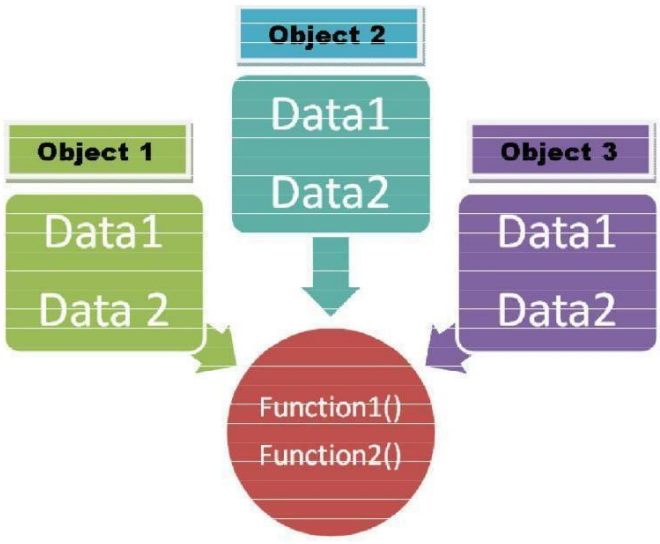
Properties of inline function:

1.Inline function sends request but not a command to compiler
2.Compiler my serve or ignore the request
3.if function has too many lines of code or if it has complicated logic then it is executed as normal function

Situations where inline does not work:

- A function that is returning value , if it contains switch ,loop or both then it is treated as normal function.
- if a function is not returning any value and it contains a return statement then it is treated as normal function
- If function contains static variables then it is executed as normal function
- If the inline function is declared as recursive function then it is executed as normal function.

Memory Allocation for Objects: Memory for objects is allocated when they are declared but not when class is defined. All objects in a given class uses same member functions. The member functions are created and placed in memory only once when they are defined in class definition



STATIC CLASS MEMBERS

Static Data Members

Static Member Functions

Static Data Members:

A data member of a class can be qualified as static. A static member variable has certain special characteristics:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- 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.
- It is visible only within the class, but its lifetime is the entire program.
- Static data member is defined by keyword **„static"**

Syntax:

Data type class name::static_variable Name;

Ex: int item::count;

```
#include<iostream.h>
#include<conio.h>
class item
{
        static int count;
        int number;
public:
        void getdata(int a)
        {
                number=a;
                count++;
        }
        void getcount()
        {
        cout<<"count is"<<count;
        }
};
int item::count;//decleration
int main()
{
item a,b,c;
        a.getcount();
        b.getcount();
        c.getcount();
        a.getdata(100);
        b.getdata(200);
        c.getdata(300);
        cout<<"After reading data";
        a.getcount();
        b.getcount();
        c.getcount();
        return 0;
}
```

Output:

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

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

A static member function is to 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()
        {
                code=++count;
        }
        void showcode()
        {
                cout<<"object number"<<code;
        }
        static void showcount()
        {
                cout<<"count"<<count;
        }
};
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.showcode();
        return 0;
}
```

Output:

```
count 2
count 3
object number 1
object number 2
object number 3
```

Arrays of Objects: Arrays of variables of type "class" is known as "Array of objects". An array of objects is stored inside the memory in the same way as in an ordinary array.

Syntax:

```
        class class_name
        {

        private:
                data_type members;
        public:
```

```
        data_type members;
        member functions;
    };
```

Array of objects:

Class_name object_name[size];
Where size is the size of array
Ex:
Myclass obj[10];

Write a program to initialize array of objects and print them

```
#include<iostream>
using namespace std;
class MyClass
{
int a;
public:
    void set(int x)
    {
        a=x;
    }
    int get()
    {
        return a;
    }
};
int main()
{
MyClass obj[5];
for(int i=0;i<5;i++)
obj[i].set(i);
for(int i=0;i<5;i++)
 cout<<"obj["<<i<<"].get():"<<obj[i].get()<<endl;
}
```

Output:

```
obj[0].get():0
obj[1].get():1
obj[2].get():2
obj[3].get():3
obj[4].get():4
```

Objects as Function Arguments: Objects can be used as arguments to functions This can be done in three ways

a. Pass-by-value or call by value
b. Pass-by-address or call by address
c. Pass-by-reference or call by reference

a.Pass-by-value – A copy of object (actual object) is sent to function and assigned to the object of called function (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object. write a program to swap values of two objects

write a program to swap values of two objects

```
#include<iostream.h>
using namespace std;
class sample2;
class sample1
{
int a;
public:
```

```
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 x,sample2 y);
};
void sample1::getdata(int x)
{
        a=x;
}
class sample2
{
int b;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);

friend void swap(sample1 x,sample2 y);
};
void sample2::getdata(int x)
{
b=x;
}
void display(sample1 x,sample2 y)
{
        cout<<"Data in object 1 is"<<endl;
        cout<<"a="<<x.a<<endl;
        cout<<"Data in object 2 is"<<endl;
        cout<<"b="<<y.b<<endl;
}
void swap(sample1 x,sample2 y)
{
int t;
        t=x.a;
        x.a=y.b;
        y.b=t;
}
int main()
{
sample1 obj1;
sample2 obj2;
        obj1.getdata(5);
        obj2.getdata(15);
        cout<<"Before Swap of data between Two objects\n
        "; display(obj1,obj2);
        swap(obj1,obj2);
        cout<<"after Swap of data between Two objects\n ";
        display(obj1,obj2);
}
```

```
Before Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
```

b. Pass-by-address: Address of the object is sent as argument to function.

Here ampersand(&) is used as address operator and arrow (->) is used as de referencing operator. If any change made to formal arguments then there is a change to actual arguments

write a program to swap values of two objects

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

```
using namespace std;
class sample2;
class sample1
{
int a;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 *x,sample2 *y);

};
void sample1::getdata(int x)
{
a=x;
}
class sample2
{
int b;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 *x,sample2 *y);
};
void sample2::getdata(int x)
{
        b=x;
}
void display(sample1 x,sample2 y)
{
        cout<<"Data in object 1 is"<<endl;
        cout<<"a="<<x.a<<endl;
        cout<<"Data in object 2 is"<<endl;
        cout<<"b="<<y.b<<endl;
}
void swap(sample1 *x,sample2 *y)
{
int t;
        t=x->a;
        x->a=y->b;
        y->b=t;
}
int main()
{
sample1 obj1;
sample2 obj2;
        obj1.getdata(5);
        obj2.getdata(15);
        cout<<"Before Swap of data between Two objects\n ";
        display(obj1,obj2);
        swap(&obj1,&obj2);
        cout<<"after Swap of data between Two objects\n ";
        display(obj1,obj2);
}
```

```
Before Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
 Data in object 1 is a=15
```

Data in object 2 is b=5

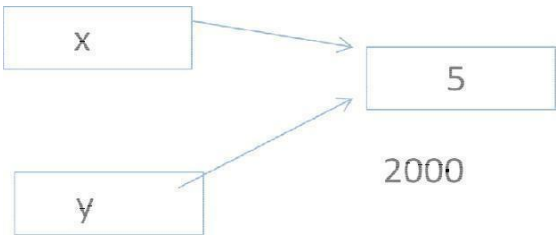
c.Pass-by-reference:A reference of object is sent as argument to function.
Reference to a variable provides alternate name for previously defined variable. If any change made to reference variable then there is a change to original variable.

A reference variable can be declared as follows

Datatype & reference variable =variable;

Ex:

```
int x=5;
int &y=x;
```



Write a program to find sum of n natural numbers using reference variable

```
#include<iostream.h>
using namespace std;
int main()
{
int i=0;
int &j=i;
int s=0;
int n;
cout<<"Enter n:";
cin>>n;
while(j<=n)
{
s=s+i;
i++;
}
cout<<"sum="<<s<<endl;
}
```

Output:

```
Enter n:10
sum=55
```

write a program to swap values of two objects

```
#include<iostream.h>
using namespace std;
class sample2;
class sample1
{
int a;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 &x,sample2 &y);
};
```

```
void sample1::getdata(int x)
{
a=x;
}
class sample2
{
int b;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 &x,sample2 &y);
};
void sample2::getdata(int x)
{
b=x;
}
void display(sample1 x,sample2 y)
{
        cout<<"Data in object 1 is"<<endl;
        cout<<"a="<<x.a<<endl;
        cout<<"Data in object 2 is"<<endl;
        cout<<"b="<<y.b<<endl;
}
void swap(sample1 &x,sample2 &y)
{
int t;
        t=x.a;
        x.a=y.b;
        y.b=t;
}
int main()
{
sample1 obj1;
sample2 obj2;
        obj1.getdata(5);
        obj2.getdata(15);
        cout<<"Before Swap of data between Two objects\n ";
        display(obj1,obj2);
        swap(obj1,obj2);
        cout<<"after Swap of data between Two objects\n ";
        display(obj1,obj2);
}
```

Output:

```
Before Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
 Data in object 1 is a=15
Data in object 2 is b=5
```

FRIEND FUNCTIONS: The private members cannot be accessed from outside the class. i.e.… a non member function cannot have an access to the private data of a class. In C++ a non member function can access private by making the function friendly to a class.

Definition:

A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator for defining . It can access private members of a class. It is declared by using keyword **"friend"**

Ex:

```
class sample
{
int x,y;
public:
        sample(int a,int b);
friend int sum(sample s);
};
sample::sample(int a,int b)
{
x=a;y=b;
}
int sum(samples s)
{
int sum;
        sum=s.x+s.y;
        return 0;
}
void main()
{
Sample obj(2,3);
int res=sum(obj);
cout<< "sum="<<res<<endl;
}
```

A friend function possesses certain special characteristics:

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

```
#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()
{
sample X;
        X.setvalue();
        cout<<"Mean value="<<mean(X);
        return 0;
}
```

write a program to find max of two numbers using friend function for two different classes

```
#include<iostream>
using namespace std;
class sample2;
class sample1
{
int x;
public:
        sample1(int a);
        friend void max(sample1 s1,sample2 s2)
};
sample1::sample1(int a)
{
x=a;
}
class sample2
{
int y;
public:
        sample2(int b);
        friend void max(sample1 s1,sample2 s2)
};
Sample2::sample2(int b)
{
        y=b;
}
void max(sample1 s1,sample2 s2)
{
If(s1.x>s2.y)
        cout<<"Data member in Object of class sample1 is larger "<<endl;
else
        cout<<"Data member in Object of class sample2 is larger "<<endl;
}
void main()
{
sample1 obj1(3);
sample2 obj2(5);
max(obj1,obj2);
}
```

Write a program to add complex numbers using friend function

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

```
{
float real,img;
public:
        complex();
        complex(float x,float y)
        friend complex add_complex(complex c1,complex c2);
};
complex::complex()
{
        real=img=0;
}
complex::complex(float x,float y)
{
        real=x;img=y;
}
complex add_complex(complex c1,complex c2)
{
complex t;
        t.real=c1.real+c2.real;
        t.img=c1.img+c2.img;
        return t;
}
void complex::display ()
{
        if(img<0)
        {img=-img;
                cout<<real<<"-i"<<img<<endl
        }
        else
        {
                cout<<real<<"+i"<<img<<endl
        }
}
int main()
{
complex obj1(2,3);
complex obj2(-4,-6);
        complex obj3=add_compex(obj1,obj2);
        obj3.display();
        return 0;
}
```

Friend Class: A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

```
#include <iostream.h>
 class sample_1
{
        friend class sample_2;//declaring friend class
        int a,b;
public:
        void getdata_1()
        {
                cout<<"Enter A & B values in class sample_1";
                cin>>a>>b;
        }
```

```
        void display_1()
        {
                cout<<"A="<<a<<endl;
                cout<<"B="<<b<<endl;
        }
};
class sample_2
{
        int c,d,sum;
        sample_1 obj1;
public:
        void getdata_2()
        {
                obj1.getdata_1();
                cout<<"Enter C & D values in class sample_2";
                cin>>c>>d;
        }
        void sum_2()
        {
                sum=obj1.a+obj1.b+c+d;
        }

        void display_2()
        {
                cout<<"A="<<obj1.a<<endl;
                cout<<"B="<<obj1.b<<endl;
                cout<<"C="<<c<<endl;
                cout<<"D="<<d<<endl;
                cout<<"SUM="<<sum<<endl;
        }
};
int main()
{
sample_1 s1;
        s1.getdata_1();
        s1.display_1();
sample_2 s2;
        s2.getdata_2();
        s2.sum_2();
        s2.display_2();
}
```

```
Enter A & B values in class sample_1:1 2
A=1
B=2
Enter A & B values in class sample_1:1 2 3 4
Enter C & D values in class sample_2:A=1
B=2
C=3
D=4
SUM=10
```

CONSTRUCTORS AND DESTRUCTORS

Constructor is a 'special' member function whose task is to initialize the object of it's class. A ***constructor*** is a special method that creates and initializes an object of a particular class. It has the same name as its class and may accept arguments. In this respect, it is similar to any other function.

If you do not explicitly declare a constructor for a class, the C++ compiler automatically generates a **default constructor that has no arguments**.

A constructor is declared and defined as follows:

```
// class with a constructor
class integer
{
        int m,n;
public:
        integer(void);              // constructor declared
        …….
};
integer : : integer(void)           //constructor defined
{
        m=0; n=0;
}
```

The constructor functions have some special characteristics, These are :

- **They should be declared in the public section.**
- **They are invoked automatically when the objects are created.**
- **They do not have return values, not even void and therefore, and they cannot return values.**
- **They cannot be inherited, though a derived class can call the base class constructor.**
- **Like other C++ functions, they can have default arguments.**
- **Constructors cannot be virtual.**
- **We cannot refer to their addresses.**
- **An object with a constructor (or destructor) cannot be used as a member or union.**
- **They make 'implicit calls' to the operators new and delete when memory allocation is required.**

PARAMETERIZED CONSTRUCTORS

The constructors that can take arguments are called parameterized constructors. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly
- By calling the constructor implicitly

```
Example:    1. Integer int1 = Integer(0,100);    //Explicit call
            2. Integer int1(0,100)               //Implicit call
```

CONSTRUCTOR OVERLOADING (or) MULTIPLE CONSTRUCTORS IN A CLASS

C++ Permits us to use more than one constructor in the same class. When more than one constructor function is defined in a class, we say that the constructor is overloaded.

CONSTRCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments.

```
Example:    class complex
            {
                    float x, y;
              public:
                  complex( float real, float imag=0 ); //Constructor with default argument
                    ……
            }
```

DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during runtime.

COPY CONSTRUCTOR

A copy constructor is used to declare and initialize an object from another object. A copy constructor takes a reference to an object of the same class as itself as an argument.
For example, the statement:

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1.
Another form of this statement is

```
integer I1 = I1;
```

The process of initializing through a copy constructor is known as copy initialization.

DYNAMIC CONSTRUCTOR

Allocation of memory to objects at the time of their construction is known as Dynamic construction of objects. The memory is allocated with the help of the new operator.

```
/* Example program for dynamic constructor and
   Destructor */
#include <iostream.h>
#include <string.h>
class String
{
        char *name;
        int length;
 public:
        String(char *s)
        {
                length = strlen(s);
                name = new char[length + 1];
                strcpy( name, s );
        }
        void display(void)
        {
                cout << name << "\n";
        }
        ~String( );
};
```

```
String :: ~String( )
{
        delete name;
}

void main( )
{
        String name1("Vinayaga ");
        String name2("College ");
        name1.display( );
        name2.display( );
}
```


DESTRUCTORS:

A ***destructor***, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a **tilde(~).** For example , the destructor for the class String can be defined as shown below:

~String( ) { }

A destructor never takes any argument nor does it return any value. It will be invoked implicitly be the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.

const OBJECTS

We may create and use constant objects using **const** keyword before object declaration.

```
Example:        const Matrix  X(m,n);
```

Any attempt to modify the values of m and n will generate compile-time error.

```
/* Program using class, object, member function, constructors and Destructors for calculating area and perimeter of a circle */
#include <iostream.h>
#include <conio.h>
class circle
{
private:
        float radius, area, perimeter;
public:
        circle() {}         //Default constructor
        circle(float r)     //Parameterized Constructor
        {
                radius = r;
                area = 0;
                perimeter = 0;
        }
        circle(float a, float p, float r=25)
                // Constructor with Default Argument
        {
                radius = r;
                area = a;
                perimeter=p;
        }
        circle(circle & x) //Copy constructor
        {
                radius = x.radius;
                area     = x.area;
                perimeter= x.perimeter;
        }

        void read(); //Member function1 Declaration
       // Member function 2 defined inside the class
        void display()
        {
        cout<<"\n\n Given radius is :";
        cout<<radius;
        cout<<"\n Area of the circle is :";
        cout<<3.14 * radius * radius;
        cout<<"\n Perimeter of the circle is :";
        cout<<2 * 3.14 * radius << endl;
        }
        ~circle() {}
        //Destructor
};
       //Member function1 defined outside the class
void circle::read()
{
        cout<<"\n Enter the radius of circle :";
        cin>>radius;
}
```

```
int main()
{
        clrscr();
        cout<<"\n Program for calculating area and
                perimeter of a circle";
        cout<<"\n ------------------------------------ ";
        float n;

        circle C1;

        circle C3(0,0);

        circle C4[10];

        C1.read();
        cout << "\n ** Default Constructor **\n";
        C1.display();    //

        cout<<"\n ** Parameterized Constructor and
                Dynamic Initialization of Objects **";
        cout<<"\n Enter radius value :";
        cin>>n;
        circle C2 = circle(n);
        C2.display();

        cout<<"\n ** Constructor with Default
                        arguments **";
        C3.display();

        cout<<"\n ***** Array of Objects *****";
        for(int i=0;i<2;i++)
        {
          C4[i].read();
          C4[i].display();
        }

        circle C5 = C1;
        circle C6(C2);

        cout<<"\n ***** Copy Constructor *****";
        C5.display();
        C6.display();
        getch();
        return 0;
}
```

This can be used to uncover a hidden variable. It take the following form

:: variable-name

This operator allows access to the global version of variable. ::count means the global version of the variable count.

Example1:

```
#include <iostream>
int m=10;                    //global m
int main()
{
int m=20;                    // m redeclared, local to main
        {
            int k = m;
            int m = 30;   //m declared again
                          // local to inner block
                cout<<" we are in inner block \n";
                cout<<"k ="<<K<<"\n";
                cout<<"m ="<<m<<"\n";
                cout<<"::m ="<<::m<<"\n";
            }
            cout<<"\n we are in outer block\n");
            cout<<"m="<<m<<"\n"};
            cout<<"::m="<<::m<<"\n"};
            return 0;
}
```

Output:

We are in inner block
k = 20
m = 30
::m = 10
We are in outer block
m = 20
::m = 10

Member Dereferencing Operators:

::*---------- → To declare a pointer to a member of a class
*---------- → To access a member using object name and a pointer to that the member
->*---------- → to access a member using a pointer to the object and a pointer to that member

Memory Management Operators:

We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ support these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier work. These operators manipulate memory on free store; they are also known as ***free store*** operators.

An object can be created by using **new**, and destroyed by using **delete**. The **new** operator can be used to create objects of any type. It take the following general form

pointer-variable = **new** data-type;

Here, ***pointer-variable*** is a pointer of type ***data-type***. The **new** operator allocates sufficient memory to hold a data object of data-type and returns the address of the object. The ***pointer-variable*** holds the address of the memory space allocated. The declaration of pointer and their assignments as follows.

```
int *p =new int;
float *q = new float;
```

When a data object is no longer needed, it is destroyed to release the memory space reuse. The general form of its use is

delete pointer-varaible;

the ***pointer-variable*** is the pointer that points to a data object created with **new**. Example

```
delete p;
delete q;
```

The **new** operator offers the following advantages:

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a typecast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

Program for Dynamic Memory allocation in C++

```
#include <iostream>
using namespace std;
int main()
{int size;
int *ptr;
cout<< "Enter size of array"<<endl;
cin>> size;
ptr = new int [size];
cout<< "Enter No."<<endl;
for(int i=0;i<size;i++)
{ cin>> ptr[i]; }
cout<< "The values are"<<endl;
for(int i=0; i<size; i++)
{    cout<< ptr[i]<<",";     }
delete [] ptr;
return 0;
}
```