→ Constructors :-

Constructors are special member function, which have same name as class and they don't have any return type not even void.

They don't need object to invoke them, they invoked automatically when an object of that class is made. If a class has ~~two~~ three object that means construct will be invoked 3 times when objects are made.

Constructors can come handy, if you wish to do some work but don't want an extra function for that.

Constructors have two types.-

1) default Constructor.
   -> Don't take any argument.

(ii) Paramatrize Constructor
-> It taks argument.

To pass argument in Constructors you can pass it with object.
ex- Complex O1(3,5);

demonstration program :-

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

class sum
{
    int a,b;
    Public:

      Sum(void);
     Void calsum(Void);
};

   Sum :: Sum()
   {
      Cout<<"Input two integers whose
```

```
Sum want to print" <<endl;
Cin>> a >>b;
}

Void sum :: calsum ()
{
   Cout<<"sum of"<<a<<"and"<<b<<" is"
       << a+b <<endl;
}

int main()
{

   Sum 01;
   01.Calsum();

   return 0;
}
```

NOTE!- Constructors must be declared in Public section of class.

and,

no one can refer to the

address of a Constructors.

Paramatrize Constructors can accept arguments in two ways:-

i) implicitly:-

<class name> <object>(arguments);

ii) explicitly:-

<class name> <object> = <class name> (arguments);

Q. WAP to Creat a function which takes 2 ~~arguments~~ point objects and computes the distance b/w them.

```
#include<iostream>
#include<math.h>
using namespace std;

class point
```

```
class point
{
    int x,y;
    Public:
    point(int a, int b);
    {
        x=a;
        y=b;
    }
    friend void distance(point, point);
};

Void distance(point O1, point O2)
{
    int a= O2.x - O1.x;
    int b= O2.y - O1.y;

    cout<<"distance blw given points
        is" << sqrt(pow(a,2) + Pow(b,2))
            <<endl;
}

int main()
{
    Point P1(3,7), P2(9,11);
```

```
distance(P1,P2);
return 0;
}
```

→ Constructor overloading :-

If you want to make more than one constructor for a class then it is called constructor overloading, as all of them will have same name.

Compiler will know which constructor have to invoke by matching the arguments.

→ Constructor with default arguments

Constructor with default argument is same as function with default argument.

If you have a ~~constructor~~ parametrize constructor and for error handling in the case anyone dosn't pass arguments

in them, you can set default value as argument So that in such cases constructor uses default value to operate.

But, it can little unappropriate to use while constructor overloading as, if you have two constructor of same class one with one argument and one with two argument and constructor which has two argument you have set default value for one argument. Now, while initializing an object if you pass one argument with object, which constructor will be invoked by compilar, so avoid such cases you can use overloading and default arguments carefully when used simultaneously

→ Dynamic initializing of object using Constructors.

We can dynamically invoke an operator by using Constructor overloading.

Like if we are using more than one Constructor for one class, we can invoke operates by using different arguments as parameters.

If we give different parameters then different operaters will invoked.

While doing this we need an operater Constructor which don't take any argument and don't do anything, just become empty.

If you are using an paramatrize Constructor you must need a default Constructor which don't take any argument or do anything.

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

Class BankDeposit
{
    int principal;
    int years;
    float interestRate;
    float returnValue;

  Public:

      BankDeposit () {} //-> empty constructor
      BankDeposit(int P, int y, float r);
      BankDeposit(int P, int y, int r);
      void show();
};

BankDeposit :: BankDeposit(int P, int y, float r)
{
    principal = P;
     years = y;
     interestRate = r;
     returnValue = principle;
```

```
for(int i=0; i<y; i++)
{
  returnvalue = returnValue *(1+interest
                               Rate);
}
}

BankDeposite :: BankDeposite(int p, int y,
                             int r)
{
    principal = p;
    years = y;
    interestRate = float (r)/100;
    returnValue = principal;
  for(int i = 0; i<y ; i++)
  {
    returnValue = returnValue *
                  (1+interestRate);
  }
}
```

```
Void BankDeposite :: show(){
    cout<<endl<<" principal amount was"<<principal<<". Return value after "<<years<<" years is"<<returnValue<<endl;
}

int main()
{
    BankDeposite bd1, bd2, bd3;
    int P, y, R;
    float r;

    cout<<"Enter the value of P, y and r"<<endl;

    cin>>P>>y>>r;

    bd1= BankDeposite(P, y, r);
    bd1.show();
```

```
cout<<"\n enter the value of P, y and
      R"<<endl;

cin>>p>>y>>R;

bd2 = BankDeposit(P, y, R);
 bd2.show();

 return 0;

}
```

→ Copy Constructors :-

Copy Constructors are used when you want to copy an object in other object. Since, this Constructor copies one object to other object it is called copy constructor.

We can copy one object to other object by passing older object as argument for new object.

We can define copy constructor which takes reference of an Object and do want we want to do along with coping previous object.

But if we not define that copy constructor, compiler provides its own defination of copy constructor, which ~~cop~~ only copies value of previous operator

nothing more.

Another way of using copy constructor is, while creating/ initializing your object assign it with previous object which you want to make copy of.

But if a object is already initialized, you can't assign it with other object.

ex. –

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

class Number
{
  int a;
  Public:
    Number()
    {  a=0;
    }
```

```
Number(int num)
{
    a = num;
}

Number(Number &o1)
{
  cout<<" copy constructor called "<<endl;
    a=obj.a;
}

Void display()
{
   cout<<" The number of this
          object is "<< a <<endl;

}
};

int main()
{
   Number x, y, z(45), z2;

   x.display();
```

```
y.display();
z.display();

Number z1(z); //copy constructor
z1.display(); //will be invoked

z2=z; //copy constructor will not be
z2.display(); //invoked as already
              //initialized.

Number z3=z; //copy constructor
z3.display(); //will be invoked

return 0;

}
```

→ Destructor :-

Destructor is used by compiler to destruct the memory allocated by object, when their is no use of object or compiler come out of a block then also object initialized inside the block get destructed.

Objects initialized inside a block have scope only inside that block, hence if compiler come out of that block then they get destructed.

If you not defined a destructor in your program, compiler automatically call destructor at the end of programme but when your programme gets complicated or you are using dynamic memory allocation you must need more control on memory, so, you need

destructor there.

Also destructor neither returns any value nor take any arguments.

Syntax :-

```
~<class name>
{
    //code;
}
```

block is nothing but lines of code incapsulated inside curly braces ({}).

Example programme :-

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

class num
{
```

```
static int count;
public:
    num()
    {
        count++;
        Cout<<" At this time constructor
                is called for object
                number"<<count<<endl;
    }

    ~num()
    {
        cout<<" At this time destructor
                is called for object number"
                <<count<<endl;

        count--;
    }

};

int num::count;
int main()
{
  Cout<<" we are inside main" <<endl;
```

```
cout<<"creating first object n1"<<endl;
num n1;
{
    cout<<"entering this block"<<endl;
    cout<<"creating 2 more objects"<<endl;
    num n2,n3;
    cout<<"exiting this block"<<endl;
}

cout<<"back to main"<<endl;
return 0;
}
```