Ch : 2

Class and Objects,

Constructors and Destructors

C Structures :

- Structures allows us to create complex user defined data type that can represent an entity with different other data types.
- After declaring the structure, you can create variables of your structure to use it.
- Example,

```
struct book
{
        int book_id;
        char name[20];
        float price;
}
```

Limitations of Structure :

- You can perform all the normal operations on the structure members but you cannot treat structure variables as normal variables
- struct number

```
{
        int a;
        float b;
 };
 struct number n1,n2,n3;
  n1=n2+n3;
```

Specifying Class :

- We can create a class by declaring its member variables and member functions
- The member functions have to be defined after declaration
- The class is also known as Abstract Data Type (ADT) because you will create its variables(objects) similar to other data types after creating class
- Class specification includes class declaration and member function definition

Cont...

```
Class class_nm
{
        private:
                datatype var1;
                datatype var2;
                ……..
                returntype fun_nm(args);
                returntype fun_nm(args);
                ……
        public:
                datatype var1;
                datatype var2;
                ……..
                returntype fun_nm(args);
                returntype fun_nm(args);
                ……
};
```

Example :

```
Class student
{
        int roll_no;
        char name[20];
        float per;
    public:
        void input();
        void display();
};
```

Defining member functions :

- We have to define each member functions declared in class
- The functions can be defined at two types :
- Inside the class
- Outside the class
- Wherever we define, it will work same way, it will not make any differences on working of functions

Defining member function inside the class :

```
#include<iostream.h>
#include<conio.h>
Class book
{
        int id;
        char nm[20];
        float price;
        public:
                void input()
                {
                  cout<<“enter book id”;
                  cin>>id;
                    cout<<“enter book name”;
                cin>>nm;
                 cout<<“enter book price”;
                 cin>>price;
                 }
                 void display()
                 {
                     cout<<“book details”;
                      cout<<“book id”<<id;
                     cout<<“book name”<<nm;
                     cout<<“book price”<<price;
                 }
};
```

Cont…

```
int main()
{
        clrscr();
        book b;
        b.input();
        b.display();
        getch();
        return 0;
}
```

Defining member function outside the class :

- I we want, we can define function outside the class if we do not want to make it inside.
- Inside the class, we need to place only prototypes and outside the class, the functions are defined
- Syntax :

```
return_type class_name  ::  function_nm(args)
{
        function body
}
```

The scope resolution operator (::) is used to specify that the function is member of the class specified by class_name

Example :

```
#include<iostream.h>
#include<conio.h>
class Example
{
        public:
                void  print();

};
void Example :: print()
{
        cout<<"hello";
}
```

```
int main()
{
        clrscr():
        Example e;
        e.print();
        getch();
        return 0;
}
```

Making Outside Function Inline :

- We can also make a function inline, defined outside the class by using the keyword inline.
- Here is an example,

```
#include<iostream.h>
#include<conio.h>
Class demoinline
{
        int a;
        public:
                void getdata(int n);
                void show();
};
Inline void demoinline :: getdata(int n)
{
        a=n;
}
Inline void demoinline :: show()
{
        cout<<“value of a=“<<a;
}

Void main()
{
        clrscr();
        demoinline obj;
        obj.getdata(10);
        obj.show();
        getch();
}
```

Nesting Of Member Functions :

- If a member function calls another member function of its class, it is known as nesting of member functions
- When a function calls another member function of its own class, it does not need to use dot(.) operator to call it.
- Here is an example,

- Obj.funmn()

Example :

```
#include<iostream.h>
#include<conio.h>
Class number
{

        public:
                int a;
                void get();
                int square(int a);
                void display(a);
};
Void number :: get()
{
        cout<<“Enter number”;
        cin>>a;
}
```

```
Int number :: square (int a)
{
        return a*a;
}
Void number :: display()
{
        int s = square(a);
        cout<<“Square :”<<s;
}
Void main()
{
        number n;
        n.get();
        n.square(a);
        n.display();
        getch();
}
```

Array Within A Class :

- We can also use array as member variable of a class
- For example,
- If we need to have a group of variables as member such as marks of students, sales of months etc
- We can have array as member variable
- Here is an example,

Example :

```
#include<iostream.h>
#include<conio.h>
Class student
{
        int id;
        char name[10];
        int marks[3];
        float per;
public:
        void getdata();
        void calculate();
        void display();
};
```

```
Void student :: getdata()
{
        cout<<“Enter Rollno”;
        cin>>id;
        cout<<“Enter student name”;
        cin>>name;
        cout<<“Enter marks of 3
        subjects”;
        for(int i=0;i<3;i++)
        {
                cin>>marks[i];
        }
}
```

```
Void student :: calculate()
{
        int total=0;
        for(int i=0;i<3;i++)
        {
                total= total+marks[i];
        }
        per=total/3;
}
```

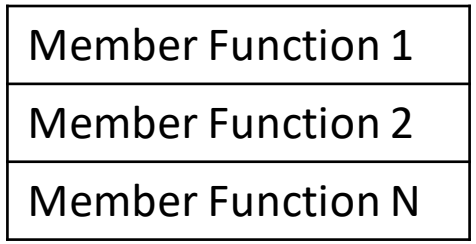
```
Void student :: display()
{
        cout<<“Student information”;
        cout<<“id =“<<id;
        cout<<“name =“<<name;
        cout<<“Percentage=“<<per;
}
```

```
Void main()
{
        clrscr();
        student s;
        s.getdata();
        s.calculate();
        s.display();
        getch();
}
```

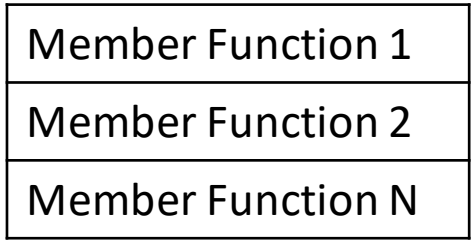
Memory Allocation Of Objects :

- When object of the class is created, memory is allocated to the object according to the member variables of the class
- But the memory space for the member function is allocated when they are defined
- So the complete memory allocation is done when an object is created
- Individual memory is allocated for each object created
- But the common memory is allocated for the member function, no separate memory space is allocated for functions

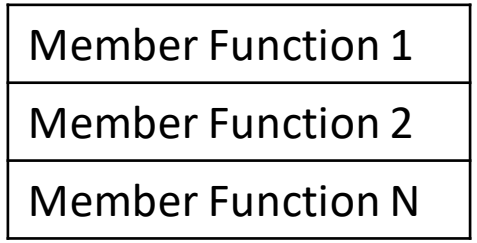
Cont..



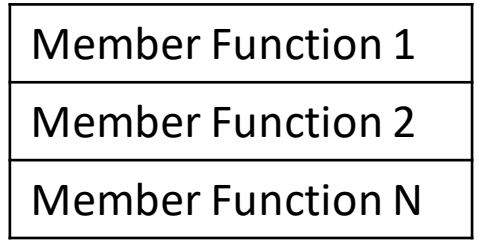
Common Memory allocated for all objects when functions are defined



Object1



Object2



Object3

Example :

```
#include<iostream.h>
#include<conio.h>
Class test
{
        int a,b;    //4
        char c[10];    //10
        float d;   //4
};
```

```
Void main()
{
        clrscr();
        test t;
        cout<<“Size of t”<<sizeof(t);
        getch();
}
```

18

Static Data Member :

- We can have variables or functions as static in our class
- Some properties of static member variables:
  - It is automatically initialized to 0 when an object is created first time
  - It cannot be initialized explicitly
  - **The value of static variable remains same for all objects**
  - It can be accessed only within the class
  - It cannot be destroyed during the program

Example :

```
#include<iostream.h>
#include<conio.h>
Class staticex
{
        int a;
        static int b;           //0
        public:
                void getval(int x)
                {
                a=x;
                b++;        //2
                cout<<a;
                }
                void getstatic()
                {
                        cout<<b;  //2
                }
};
Int staticex :: b;
Void main()
{
        staticex e1,e2;
        cout<<“static value for s1”;
        s1.getstatic();         0
        cout<<“static value for s2”;
        s2.getstatic();         0
```

Cont...

```
        cout<<“value for s1”;
        s1.getval(111);
        cout<<“value for s2”;
        s2.getval(222);
        cout<<“static value for s1”;
        s1.getstatic();
       cout<<“static value for s2”;
        s2.getstatic();
        getch();
}
```

- Static members are stored separately unlike normal variables, so

Int staticex :: b;

Is necessary.

Static Member Function :

- We can also create static member functions like member variables
- Some characteristics of it :
  - In static function, we can only use other static member variables
  - Static member functions generates same output irrespective of objects
  - It can be called by the class name using scope resolution operator instead of dot operator
  - Syntax:
    - class_nm :: static_member_function();

Example:

```
#include<iostream.h>
#include<conio.h>
Class number
{
        int a;
        static int count;
        public:
        void shownum()
        {
                count++;                //3
                a=count;
                cout<<a;                //3

        static void showcount()
        {
                cout<<count;            //3
        }
};
```

```
Int number :: count;
Void main()
{
        number n1,n2,n3;
        number :: showcount();      0
        n1.shownum();               1
        n2.shownum();               2
        n3.shownum();               3
        number::showcount();        3
        number::showcount();        3
        number::showcount();        3
        getch();
}
```

Array Of Objects :

- Like any other normal variables, you can also create an array of objects for class
- For ex,
- If you have created a class student, you can create an array of student objects to represent 50 students
- For ex,
- student s[50];
- Now to access the member function of the class, you can use array index as:
- s[5].display();
- s[10].getdata();

Cont…

- or we can use loop

```
For(int i=0; i<60; i++)
{
    s[i].getdata();
    s[i].display();
}
```

```
s[0].getdata();
s[0].display();
```

Example :

```
#include<iostream.h>
#include<conio.h>
Class book
{
        int id;
        char name[10];
        float price;
Public:
        void input()
        {
                cout<<“Enter book id”;
                cin>>id;
                cout<<“Enter book name”;
                cin>>name;
                cout<<“Enter book price”;
                cin>>price;
        }
void show()
        {
                cout<<“Book id is”<<id<<endl;
                cout<<“Book name is”<<name<<endl;
                cout<<“Book price is”<<price;
        }
};
```

Cont…

```
Void main()
{
        book b[3];
        for(int i=0;i<3;i++)
        {
                b[i].input();
        }
        cout<<“Book information”;
        for(i=0;i<3;i++)
        {
                b[i].show();
        }
        getch();
}
```

Object As Function Arguments :

- We know that, member functions can have arguments, just like any other normal variables, we can also pass objects as function arguments
- As the objects are of type **class**, we have to specify the class name as the type of object arguments **student s;**
- The concept, call by value and call by reference applies to the function having object as arguments
- If you pass address of the object to the function, is call by reference, so any changes made on the object will also affect the object values

Cont…

- But if you pass object normally, it is call by value. So any changes made on the object will not reflect to the original object
- Here is an example,

Example :

```
#include<iostream.h>
#include<conio.h>
Class numbers
{
        int a,b;
Public:
        void input(int x,int y)
        {
                a=x;    100     10
                b=y;    200     20
        }
};
```

```
void sum(numbers n1, numbers n2)
{
        a=n1.a+n2.a;
        b=n1.b+n2.b;
}
void output()
{
        cout<<a<<b;
}
```

Cont…

```
Void main()
{
        numbers n1,n2,n3;
        n1.input(100,200);
        n2.input(10,20);
        n3.sum(n1,n2);
        n1.output();
        n2.output();
        n3.output();
        getch();
}
```

Friend Function :

- Normally, the private members cannot be accessed by external functions
- Means a function which is not a member function of the class cannot have access to the private members of the class
- C++ introduces a kind of functions known as friend functions which behaves like friend of the class
- The friend functions have access to the private members of the class
- You can define a function friendly to one or more classes allowing the function to access the public as well as private members of all the classes to which it is declared as friend

Cont…

- To declare a friend function, **friend** keyword is used. Following is the general form :

```
Class Test
{
        private:
                member variables…
        public:
                member functions….
                friend void abc(args); //friend function declaration
};
```

Cont…

- To understand friend function more clearly,
- The friend function can be declared in either public or private section of class.
- The friend function is not in the scope of the class in which is declared
- It is declared inside the class definition but it must be defined outside the class without using ::
- It is called without using object. Like, **test();**
- It will need object to access the member variables. It cannot access member variables directly like other member functions
- Generally it takes objects as arguments so that it can access member variables of the class using the object

Example :

```
#include<iostream.h>
#include<conio.h>
Class friendex
{
        int a,b;
Public:
        void setvalue(int x, int y)
        {
                a=x;
                b=y;
        }
        void display()
        {
        cout<<a<<endl<<b;
        }
        friend void sum(friendex f);
};
        void sum(friendex f)
        {
                int s=f.a+f.b;
                cout<<“sum=“<<s;
        }
Void main()
{
        friendex f;
        f.setvalue(100,200);
        f.display();
        sum(f);
        getch();
}
```

Returning Objects :

- As a member function can take objects as arguments, a function can also return objects
- As any other normal data types, a function can also return objects
- A function should specify the class name as the return type
- Consider the example :

Example :

```
#include<iostream.h>
#include<conio.h>
Class time
{
        int h,m;
        public:
        void settime(int hrs, int mnt)
        {
                h=hrs;
                m=mnt;
        }
        void display()
        {
                cout<<“Hours”<<h;
                cout<<“Min”<<m;
        }
        friend time add(time t1, time t2);
};
Time add(time t1, time t2)
{
        time t;
        t.m=t1.m+t2.m;
        t.h=t.m/60;
```

```
        t.m=t.m%60;
        t.h=t.h+t1.h+t2.h;
        return t;
}
Void main()
{
        time t1,t2,t3;
        t1.settime(1,30);
        t2.settime(2,40);
        t3=add(t1,t2);
        t1.display();
        t2.display();
        t3.display();
        getch();
}
```

Const Member Function :

- We can specify a member function by const keyword if you do not want to allow the function to modify any member variables
- A member function can be declared as const by simply adding const keyword after the function name in function declaration as well as function definition
- void test (int a, int b) const; //declaration
- void test(int a, int b) const //definition

```
{
        //code
}
```

Now if you try to change the member variables in this function, you will get error message

Pointer To Members :

- Like normal variables, you can also create pointer to member variables also
- For normal variables, you can create pointer using * operator and the address of a variable can be obtained by applying & operator
- In case of member variables, you can create pointer using ::* operator and the address of a variable can be got using & operator after the var name followed by class name and ::
- For example,
- int a =10;
- int *p=&a;

Cont…

- Same way,

```
Class test
{
        int a;
        ……
        ……
};
```

Pointer can be created by :

```
int *p=&a;
int test ::*p=&test::a;
```

Cont…

- If you have created pointer to object, you can access the member by using arrow sign instead of dot
- test t;

Test *p=&t;

To access variable using t,

t.a=10;

To access variable using p,

T->a=10;

Example :

```
Class pointer
{
        int a,b;
        public:
        void set(int p,int q)
        {
                a=p;        //50
                b=q;        //20
        }
        void display()
        {
                cout<<a<<endl<<b;
        }
        friend void add(pointer p);
        friend void sub(pointer p);
};
void add(pointer p)
{
        int pointer ::*p1=&pointer::a;        //int *p1=&a;
        int pointer ::*p2=&pointer::b;        //int *p2=&b;
        int s=p.*p1 + p.*p2;
        cout<<"Addition is"<<s;
}
Void sub(pointer p)
{
        int pointer ::*p1=&pointer::a;
        int pointer ::*p2=&pointer::b;
        pointer *ptr=&p;
        int s=ptr->*p1 – ptr->*p2;
        cout<<"Substration is"<<s;
}
```

Cont….

```
Void main()
{
        clrscr();
        pointer p;
        p.set(50,20);
        void(pointer
::*disp())=&pointer::display;
        add(p);
        sub(p);
        getch();
}
```

Local Class :

- We can create a class inside a function definition
- These types of classes are known as local classes
- For example, consider this :

```
Void abc()
{
        class test      //local class
        {
                //class definition
        };
}
```

Example :

```
#include<iostream.h>
#include<conio.h>
Class test
{
        public:
        void demo();
};

Void test :: demo()
{
        class xyz
        {
                public:
                void show()
                {
                cout<<“This is show”;
                }
        };
Cout<<“calling function”;
Xyz x;
x.show();
}
```

Cont…

```
Void main()
{
        clrscr();
        test t;
        t.demo();
        getch();
}
```

Nested Classes :

- We can create nested class by defining class inside the another class
- The class defined inside the class is known as the inner class and the class in which a class defined is known as outer class

```
Class outerclass
{
        class innerclass
        {
                //code
        };
};
```

Example :

```
#include<iostream.h>
#include<conio.h>
Class outer
{
        public:
        class inner
        {
        public:
        void showinner()
        {
        cout<<“inner”;
        }
        };
        void showouter()
        {
        cout<<“outer”;
        inner I;
        i.showinner();
        }
};
Void main()
{
        outer o1;
        o1.showouter();
        getch();
}
```

Constructors :

- Normally, functions like getdata(), input(), setvalues() are used to give values to member variables for particular objects
- In fact, in OOP, the task of initializing member variable is done by constructors.
- That is member variables can be initialized using constructors
- Constructor is a type of function that is used to construct the object of its class
- Its main task is to initialize member variables of its class so that after creating objects, you do not need to call functions mentioned above

Cont….

- Constructor is a type of member function which initialize the objects of its class
- We have mentioned the constructor as a special member function because its name as same as its class name
- You cannot give any other name to the constructors
- General form of constructor:

```
Class sample
{
        int a,b;
        public:
        sample()
        {
                a=0;
                b=0;
        }
};
```

Cont….

- Points to remember about constructors :
- Constructors are special member functions
- It has the same name as class name
- It cannot have return type, NOT EVEN VOID
- It is automatically called, we do not need to call
- **The constructor is called when the object of its class is created**
- i.e. it is implicitly called
- Constructors should be declared in public section, otherwise it cannot be accessed

Cont…

- Constructors can have default arguments
- Constructors make automatically call to **new** and **delete** for memory allocation and deallocation
- Types of Constructors:
- There are 3 types of constructors in C++ as below:
- Default Constructors (with 0 parameters)
- Parameterized Constructors (with one or more parameters)
- Copy Constructors (with objects as parameters)

Example : (Default)

```
#include<iostream.h>
#include<conio.h>
Class test
{
        int a,b;
        public:
        test()
        {
                a=0;
                b=0;
        }
        void show()
        {
                cout<<“a=“<<a<<endl;
                cout<<“b=“<<b;
                }
};

Void main()
{
        clrscr();
        test t;
        t.show();
        getch();
}
```

Parameterized Constructors :

- We can pass arguments to the constructors to initialize its objects with some specified values
- The constructors that take one or more arguments is known as parameterized constructor
- Example : class test

```
{
        int a,b;
        public:
        test(int a1, int b1)
        {
                a=a1;
                b=b1;
        }
};
```

Cont….

- Constructor is called when you create object of the class
- So you have to pass parameters to constructor at the time of creating object
- The parameterized constructors can be called by two ways:
- By making explicit call to constructor like: test t1=test(100,200);
- By making implicit call like : **test t1(100,200);**

Example :

```
#include<iostream.h>
#include<conio.h>
Class test
{
        int a,b;
        public:
        test(int x, int y)
        {
                a=x;
                b=y;
        }
        void show()
        {
                cout<<a<<endl<<b;
        }
};
```

```
Void main()
{
        clrscr();
        test t1=test(10,20);
        t1.show();
        test t2(111,222);
        t2.show();
        getch();
}
```

Multiple Constructor In A Class :

- In a program, you may need to create more than one constructor to initialize objects
- For example, we can overload constructors with different number or types of arguments similar to function overloading to create different types of objects
- When you create object with no arguments, the default constructor is called
- When you create object with number of arguments, the parameterized constructor is called

Example :

```
#include<iostream.h>
#include<conio.h>
Class box
{
        double height, width, depth;
        public:
        box()
        {
        height=width=depth=0;
        }
        box(double length)
        {
         height=width=depth=length;
        }
        Box(double h, double w, double d)
        {
                height=h;
                width=w;
                depth=d;
        }
        Void show()
        {
                cout<<“height”<<height;
                cout<<“Width”<<width;
                cout<<“Depth”<<depth;
        }
```

Constructor With Default Arguments :

- Like normal functions of C++, you can also set default arguments in constructors also
- The same rules are applied to the constructors for default arguments as for the functions
- The constructors will consider the default argument if no value is specified for it
- class test

```
{
        double p,r;
        int n;
        public:
        test(double p, int n, double r=0.12);
};
```

Cont….

- Here at the time of creating object of test class, if you do not specify value of r, it will consider it the default argument
- If you specify all the values, the specified value of r will be considered
- Example for this is in HOMEWORK

Copy Constructor :

- A copy constructor is a constructor which is used to create a new object from an existing object
- This type of constructor takes reference to an object as argument and initializes the member variables of its class with the values of the specified objects
- General form is :

```
class test
{
        int a; float b;
        public:
                test(test &t) { }
};
```

Cont….

- Now to call this constructor, you have to pass the object as argument from which you want to create a new object

  test t1;

  test t2(t1);

- You can also call copy constructor by following statement :

  test t3=t2;

- It works same as the above statement

Example :

```
#include<iostream.h>
#include<conio.h>
Class box
{
        float h,w,d;
        public:
        box()
        {
                h=w=d=0;
        }
box(float h1, float w1, float d1)
{
        h=h1;
        w=w1;
        d=d1;
}
box(box &b)
{
        h=b.h;
        w=b.w;
        d=b.d;
}
```

Cont…

```
        void display()
        {
                cout<<“Height”<<h;
                cout<<“Width”<<w;
                cout<<“Depth”<<d;
                cout<<“Volume”<<h*w*d;
        }
};
```

```
Void main()
{
        box b1;
        box b2(10,20,30);
        box b3(b2);
        cout<<“Box 1”;
        b1.display();
        cout<<“Box 2”;
        b2.display();
        cout<<“Box 3”;
        b3.display();
        getch();
}
```

Dynamic Initialization Of Objects :

- We can provide the initial values for an object dynamically at runtime
- This is known as the dynamic initialization of objects
- You can get the values from the user at runtime and these values can be passed to the constructor to build the object
- Consider the example, where the values are given to the constructor dynamically

Example :

```
#include<iostream.h>
#include<conio.h>
Class box
{
        float h,w,d;
        public:
        box()
        {
                h=w=d=0;
        }
box(float h1, float w1, float d1)
{
        h=h1;
        w=w1;
        d=d1;
}
box(box &b)
{
        h=b.h;
        w=b.w;
        d=b.d;
}
```

Cont…

```
        void display()
        {
        cout<<“Height”<<h;
        cout<<“Width”<<w;
        cout<<“Depth”<<d;
        cout<<“Volume”<<h*w*d;
        }
};
```

```
Void main()
{
        box b1;
        box b2(10,20,30);
        box b3(b2);
        cout<<“box 1”;
        b1.display();
        cout<<“box 2”;
        b2.display();
        cout<<“box 3”;
        b3.display();
        getch();
}
```

Dynamic Constructors :

- In the dynamic constructor the memory is allocated to the object dynamically at the time of creation of objects
- It will save the memory as only the required amount of memory is allocated to the objects
- The new operator is used to allocate memory to the objects
- Here in example, the another constructor we have allocated memory as per the size of string passed to it and additional space for null character

Example :

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
Class text
{
        char *name;
        int len;
        public:
        text()
        {
                len=0;
                name=new char[1];
        }
        text(char *n)
        {
                len=strlen(n);
                name=new char[len+1];
                strcpy(name,n);
        }
        void show()
        {
                cout<<“Name is”<<name;
        }
};
```

Cont…

```
Void main()
{
        char *name=“kscpac”;
        text t1;
        t1=text(name);
        t1.show();
        text t2(“abcd”);
        t2.show();
        getch();
}
```

MIL (Member Initialization List) :

- The MIL is a way by which you can initialize the member variables in the constructor
- The list of members to be initialized is written with constructor separated by comma (,) followed by a colon
- Syntax : **constructor(arg1,arg2,….): var1(val),var2(val),….**

```
{
        //other code
}
```

Here, in the parenthesis near variables, you can also pass expression:

Cont..

- Number (int a, int b) : x(**a+b**), y(**b*2**)

  {

  //other code

  }

- Advantages of using MIL :
- There are definitely some advantages of using MIL rather than the normal assignments we do such as :

Cont…

```
Constructor (int a, int b, int c)
{
        x=a;
        y=b;
        z=c;
}
```

(1) The MIL is more efficient than the normal assignments

(2) It actually initializes the member variables because the assignment version constructor first calls default to initialize the member variables

Cont….

(3) So all the work performed by the default constructor is wasted and done again

(4) We can also use expressions in MIL which will save code and execution time

Example :

```
#include<iostream.h>
#include<conio.h>
Class test
{
        int a,b;
        public:
        test(int x, int y):a(x),b(y)
        {
                cout<<“This is MIL”;
        }
        void display()
        {
                cout<<“a=“<<a<<endl;
                cout<<“b=“<<b;
                }
};
Void main()
{
        clrscr();
        test t(11,22);
        t.display();
        getch();
}
```

Destructors :

- A destructor is also a special kind of member function which is used to destroy the object created by constructor
- It is special because like constructor, it has also same name as the class name
- The destructor is written by specifying a tilde (~) sign before its name
- For example,

```
~ test()
{
        //code
}
```

Characteristics of Destructor :

- It has the same name as the class name
- It starts with tilde (~) sign
- It cannot take any arguments
- It does not return any value
- It is called automatically when an object goes out of scope
- It releases the memory allocated to the object by constructor
- Destructors cannot be overloaded

Importance :

- When we use constructors to create objects, it allocates memory to those objects
- Now as the new objects are created more and more memory is allocated to the objects
- At some point these constructors may not be in use i.e. in the scope but still they have occupied some memory
- In some systems the memory is very important so we have to take care about memory management
- In C++, destructors are the solution to this problem, when the object goes out of scope, destructors is called automatically and releases the memory allocated by the object

Cont..

- If in the constructor, the memory is allocated by **new** keyword, it should be deleted by **delete** keyword in destructor
- Example,

```
Test()
{
        a=new char[len+1];
}
~Test()
{
        delete a;
}
```

Example :

```
#include<iostream.h>
#include<conio.h>
Class test
{
        public:
        test()
        {
        cout<<“Object created”;
        }
        ~test()
        {
        cout<<“Object Deleted”;
        }
};
```

```
Void main()
{
        test t1,t2;
        {
        cout<<“creating object in a block”;
        test t3;
        cout<<“object will be destroyed when block ends”;
        cout<<“press any key to exit”;
        getch();
        }
        getch();
}
```