Lecture 4

CONSTRUCTORS AND DESTRUCTORS

4.1 Introduction

C++ provides a special member function called the ***constructor*** which enables an object to initialize itself when it is created. This is known as ***automatic initialization*** of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

4.2 constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. **A constructor is a member function that is executed automatically whenever an object is created.** It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it construct the values of data members of the class.

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;
}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

integer int1; //object int1 created

Not only creates the object **int1** of type integer but also **initializes** its data member's **m** and **n** to zero.

A constructor that accepts no parameters is called the ***default constructor***. The default constructor for class **A** is **A::A( ).** If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as (**A a**;)

Invokes the default constructor of the compiler to create the object **a**.

The constructor functions have some special characteristics:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, 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.
- We can be defined as **inline** function

4.3 Parameterized Constructors

The constructor **integer ( ),** defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.

```
class integer
{
        int   m , n;
   public:
      integer ( int x , int y );       // parameterized constructor
      …….
      ……
};
 integer:: integer (int x , int y )        // constructor defined
{
     m = x;  n = y;
}
```

In above program 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**.

The following declaration illustrates the first method:

```
integer int1= integer(0,150);    //explicit call
```

This statement creates an **integer** object **int1** and passes the values 0 and 150 to it.

The second is implemented as follows:

integer int1(0,150); //implicit call (*shorthand*)

// Class with Constructors //

```
#include <iostream>
using namespace std;
class integer
{
      int   m , n;
    public:
        integer ( int , int );              //constructor declared
        void display (void);
};
integer :: integer (int x , int y )        // constructor defined
                    {
                          m = x;  n = y;
                    }
void  integer :: display (void)
        {
              cout << " m = "  << m  << "\n";
              cout << " n = "  << n  << "\n";
         }
main ()
{
          integer int1(0,100);               //implicit call
          integer int2= integer(25,75);    //explicit call
          cout << "\n OBJECT1"  << "\n";
```

```
        int1.display( );
        cout << "\n OBJECT2"  << "\n";
        int2.display( );
}
```

The output of above program is:

```
        OBJECT1
        m = 0
        n= 100
       OBJECT2
       m = 25
       n= 75
```

Example:-This example, COUNTER, provides a counter variable that can be modified only through its member functions.

```
// counter.cpp
// object represents a counter variable
#include <iostream>
using namespace std;
class Counter
{ private:
int count; //count
public:
Counter (): count (0) //constructor
{ /*empty body*/ }
void inc_count() //increment count
{ count++; }
int get_count() //return count
{ return count; }
};
```

```
main()
{
Counter c1, c2;                          //define and initialize
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
c1.inc_count();                          //increment c1
c2.inc_count();                          //increment c2
c2.inc_count();                          //increment c2
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count(); cout << endl;
}
```

The Counter class has one data member: **count,** of type unsigned int (since the count is always positive).

It has **three member functions**:

1. **the constructor Counter()**, which we'll look at in a moment;
2. **inc_count(),** which adds 1 to count;
3. **get_count(),** which returns the current value of count.

Initializer List

One of the most common tasks a constructor carries out is initializing data members. In the

Counter class the constructor must initialize the count member to 0. One of the most common tasks a constructor carries out is initializing data members. In the Counter class the constructor must initialize the count member to 0. If multiple members must be initialized, count()

{count = 0; }

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

count() : count(0)

{ }

The initialization takes place following the member function declaratory but before the function body. It's preceded by a colon. The value is placed in parentheses following the member data.

If multiple members must be initialized, they're separated by commas. The result is the initializer list (sometimes called by other names, such as the member-initialization list).

some Class() : m1(7), m2(33), m2(4) ← initializer list

{ }

Why not initialize members in the body of the constructor? The reasons are complex, but have to do with the fact that members initialized in the initializer list are given a value before the constructor even starts to execute. This is important in some situations. For example, the initializer list is the only way to initialize const member data and references.

Actions more complicated than simple initialization must be carried out in the constructor body, as with ordinary functions. They're separated by commas. The result is the initialize list:

Some Class (): m1(7), m2(33), m3(4) ← initialize list

{ }

Counter Output

The main() part of this program exercises the Counter class by creating two counters, c1 and c2. It causes the counters to display their initial values, which—as arranged by the constructor—are 0. It then increments c1 once and c2 twice, and again causes the counters to display themselves (non-criminal behavior in this context). Here's the output:

c1=0

c2=0

c1=1

c2=2

Example:-

```
#include <iostream>
using namespace std;
class operations
{
float a,b,c; int ch;
public:
operations( );
void result( );
};
operations::operations( )
{
cout<<"Mathematical Operations \n";
cout<<" 1- Addition \n";
cout<<" 2- Subtraction \n";
cout<<" 3- Multiplication \n";
cout<<" 4- Division \n";
 cout<<" Please Enter your choice : \n";
 cin>>ch;
cout<<" Please Enter two Values a and b \n";
cin>>a>>b;
}
void operations::result( )
{
switch (ch)
{
case 1: c=a+b; cout<<a<<"+"<<b<<"="<<c<<endl;break;
```

```
case 2 :c=a-b;cout<<a<<"-"<<b<<"="<<c<<endl; break;
case 3: c=a*b;cout<<a<<"*"<<b<<"="<<c<<endl; break;
case 4: if(b!=0){ c=a/b;cout<<a<<"/"<<b<<"="<<c<<endl;}
      else   cout<<"the result of division is infinite ";
defult :cout<< "error choice";
}
}
void main( )
{
int key;
do
{    operations op;
op.result( );
 cout <<"to terminate program write key= 0 otherwise enter any value to
key \n";
 cin>>key;     } while (key! =0);}
```

4.4 Destructors

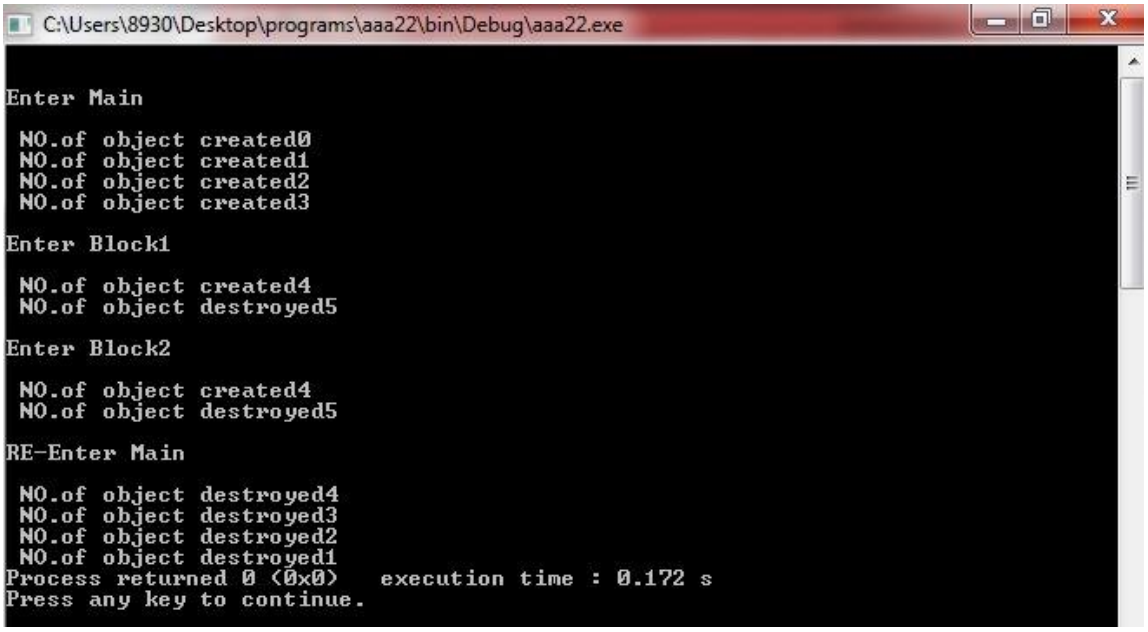
A destructor, as name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~). For example, the destructor for the class **integer** can be defined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

~integer() { }

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible.

```
// implementation of destructors //
#include <iostream>
using namespace std;
int count=0;
class alpha
{
    public:
        alpha()
        {
            count++;
            cout<< "\nNO.of object created " << count;
        }
        ~alpha()
        {
            cout<< "\nNO.of object destroyed " << count ;
            count--;
        }
};
main()
{
  cout<< "\n\nEnter Main\n;
  alpha A1,A2,A3,A4;
  {
        cout << "\n\nEnter Block1\n";
        alpha A5;
   }
   {
       cout << "\n\nEnter Block2\n";
      alpha A6;
```

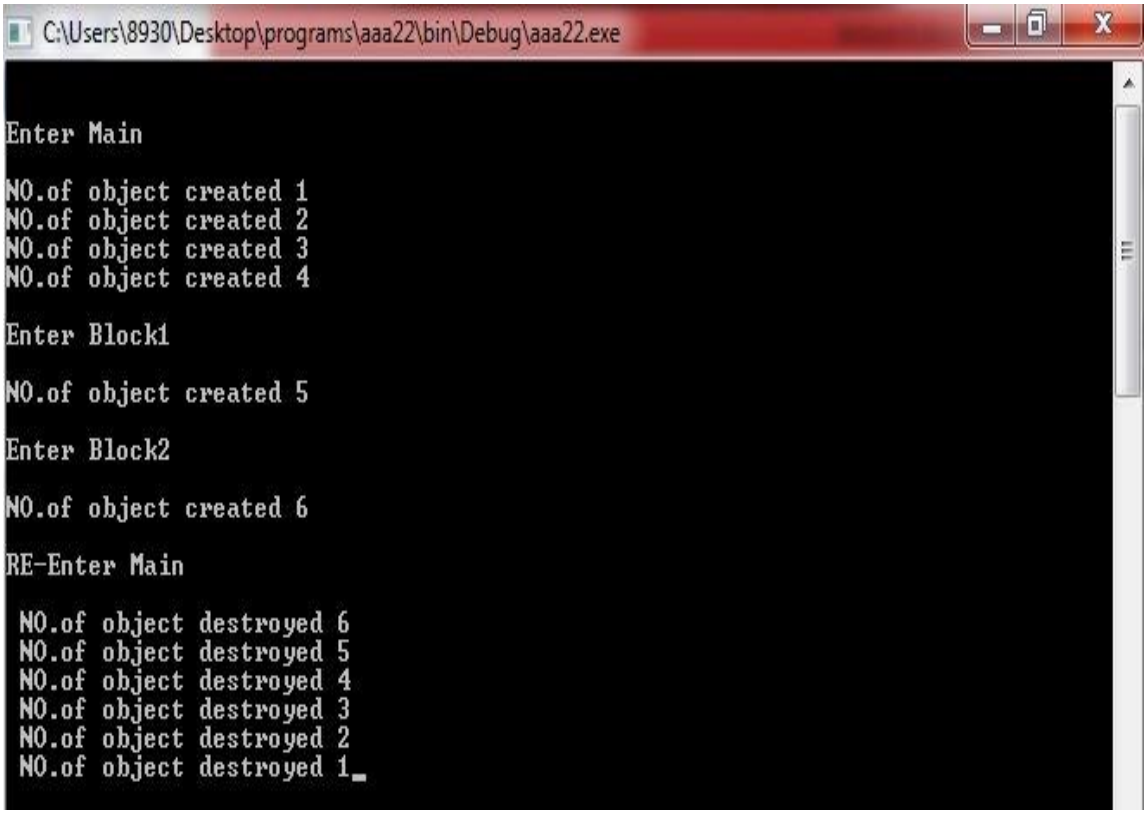
```
    }
   cout<< "\n\nRE-Enter Main\n;
}
```



الحل بطريقة ثانية

```
#include <iostream>
#include <conio.h>
using namespace std;
int count=0;
class alpha
{
    public:
        alpha()
        {
            count++;
            cout<< "\nNO.of object created " << count ;
         }
        ~alpha()
         {
```

```
            cout<< "\n NO.of object destroyed " << count ;
            count--;
        }
};
main()
{
  { cout<< "\n\nEnter Main\n";
   alpha A1,A2,A3,A4;
        cout << "\n\nEnter Block1\n";
        alpha A5;
       cout << "\n\nEnter Block2\n";
      alpha A6;
    cout<< "\n\nRE-Enter Main\n";
   }
   getch();
}
```



Note: - As the objects are created and destroyed, they increase and decrease the count. Notice that after the first group of objects is created, A5 is created, and then destroyed, A6 is created, and then destroyed. Finally, the rest of the objects are also destroyed. When the closing brace of a scope is encountered, the destructors for each object in the scope are called. Note that the objects are destroyed in the reverse order of creation.

4.5 Multiple Constructors in A Class

So far we have used two kinds of constructors. They are:

```
integer();              //No arguments
integer(int , int);     //two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In second case, the function call passes the appropriate value from **main ().** C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
{
        int m,n;
  public:
     integer()  { m=0; n=0; }    //constructor 1
     integer(int a , int b )
     {m = a ; n = b;}              //constructor 2
     integer(integer &i)
     {m =  i . m ; n = i . n;}     //constructor 3
};
```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second, receivers' two integer arguments, and the third receives one **integer** object as an argument. For example, the declaration:

integer l1;

Would automatically invoke the first constructor and the set both **m** and **n** of **l1** to zero. The statement

integer l2 (20 , 40);

Would call the second constructor, which will initialize the data members **m** and **n** of **l2** to 20 and 40 respectively. Finally, the statement

integer l3 (l2);

Would invoke the third construct which copies the value of **l2** into **l3**. That is, it sets the value of every data element of **l3** to the value of corresponding data element of **l2.** Such a constructor is called the ***copy constructor*.**

When more than one constructor function is defined in a class, we say that the constructor is overloaded.

// Overloading Constructor //

Example:- Create a class that imitates part of the functionality of the basic data type int. Call the class Int (note different capitalization). The only data in this class is an int variable. Include member functions to initialize an Int to 0, to initialize it to an int value, to display it (it looks just like an int), and to add two Int values.

Write a program that exercises this class by creating one uninitialized and two initialized Int values, adding the two initialized values and placing the response in the uninitialized value, and then displaying this result.(Instead of having z=x+y, and x,y and z are int , we could have z.add(x,y) and x,y and z are of type Int.)

Solution:-

```
// ex6_1.cpp
// uses a class to model an integer data type
#include <iostream>
using namespace std;
class Int //(not the same as int)
{
private:
int i;
public:
Int() //create an Int
{ i = 0; }
Int(int X) //create and initialize an Int
{ i = X; }
void add(Int i2, Int i3) //add two Ints
{ i = i2.i + i3.i; }
void display() //display an Int
{ cout << i; }
};
void  main()
{ Int Int1(7);                          //create and initialize an Int
Int Int2(11);                          //create and initialize an Int
Int Int3;                              //create an Int
Int3.add(Int1, Int2);                  //add two Ints
cout << "\nInt3 = "; Int3.display(); cout << endl;  }
```

4.6 Objects as Function Arguments

Our next program adds some new aspects of classes: constructor overloading, defining member functions outside the class, and perhaps most importantly objects as function arguments. Here's the listing for ENGLCON:

```
// englcon.cpp
// constructors, adds objects using member function
#include <iostream>
using namespace std;
class Distance //English Distance class
{ private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in) //constructor (two args)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << '\"';}
void Distance::add_dist(Distance d2, Distance d3)
{ inches = d2.inches + d3.inches; //add the inches
feet = 0; //(for possible carry)
```

```
if(inches >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
inches -= 12.0; //by 12.0 and
feet++; //increase feet
} //by 1
feet += d2.feet + d3.feet;
} //add the feet
};
main()
{ Distance dist1, dist3; //define two lengths
Distance dist2(11, 6.25); //define and initialize dist2
dist1.getdist(); //get dist1 from user
dist3.add_dist(dist1, dist2); //dist3 = dist1 + dist2 //display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl; }
```

This program starts with a distance dist2 set to an initial value and adds to it a distance dist1, whose value is supplied by the user, to obtain the sum of the distances. It then displays all three distances:

Enter feet: 17

Enter inches: 5.75

dist1 = 17'-5.75"

dist2 = 11'-6.25"

dist3 = 29'-0"

It's convenient to be able to give variables of type Distance a value when they are first created. That is, we would like to use definitions like:

Distance width (5, 6.25);

which defines an object, width, and simultaneously initializes it to a value of 5 for feet and 6.25 for inches. To do this we write a constructor like this:

Distance (int ft, float in): feet (ft), inches (in) { }

This sets the member data feet and inches to whatever values are passed as arguments to the constructor.

In that program there was no constructor, but our definitions worked just fine. How could they work without a constructor? Because an implicit no-argument constructor is built into the program automatically by the compiler and it's this constructor that created the objects, even though we didn't define it in the class.

This no-argument constructor is called the default constructor. Often we want to initialize data members in the default (no-argument) constructor as well. If we let the default constructor do it, we don't really know what values the data members may be given. If we care what values they may be given, we need to explicitly define the constructor. In ENGLECON we show how this looks:

Distance() : feet(0), inches(0.0) //default constructor { }

The data members are initialized to constant values, in this case the integer value 0 and the float value 0.0, for feet and inches respectively. Now we can use objects initialized with the no-argument constructor and be confident that they represent no distance (0 feet plus 0.0 inches) rather than some arbitrary value.

Since there are now two explicit constructors with the same name, Distance(), we say the constructor is overloaded. Which of the two constructors is executed when an object is created depends on how many arguments are used in the definition:

Distance length; // calls first constructor

Distance width (11, 6.0); // calls second constructor

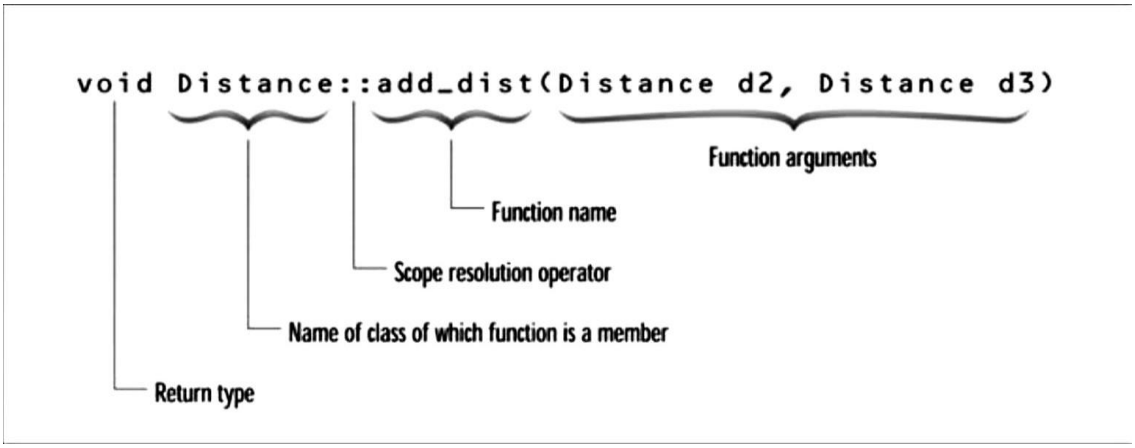
Member Functions Defined Outside the Class

In ENGLCON the add_dist() function is defined following the class definition.

//add lengths d2 and d3

```
void Distance::add_dist(Distance d2, Distance d3)
{
inches = d2.inches + d3.inches; //add the inches
feet = 0; //(for possible carry)
if(inches >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
inches -= 12.0; //by 12.0 and
feet++; //increase feet
} //by 1
feet += d2.feet + d3.feet; //add the feet
}
```

The declarator in this definition contains some unfamiliar syntax. The function name, add_dist(), is preceded by the class name, Distance, and a new symbol—the double colon (::). This symbol is called the scope resolution operator. It is a way of specifying what something is associated with.



Ex:- constructor function

```
#include<iostream>
using namespace std;
class myclass
{
int a;
public:
my class( ); // constructor function
void show( );
};
myclass::myclass( )
{
cout<<"constructor function \n";
a=10;
}
void myclass::show( )
{
cout<<a;
}
int main( )
{
myclass ob;
ob.show( );
return 0;}
```

4.7 copy constructor

A copy constructor is used to declare and initialize an object from another object. For example, the statement

integer L2(L1);

Would define the object **L2** and at the same time initialize it to the value of **L1.** Another form of this statement is

integer L2=L1;

// copy constructor //

```
#include <iostream>
using namespace std;
class code
{
      int id;
  public:
       code()  {}                    // constructor
       code (int a) {id = a;}         //constructor again
       code (code & x)               //copy constructor
        {
          id = x . id;
        }
       void display(void)
       {     cout  <<  id;    }
 };
main()
{
    code A(100);
    code B(A);
    code C=A;
    code D;
    D=A;
    cout<<"\n id of A: "; A.display();
    cout<<"\n id of B: "; B.display();
```

```
    cout<<"\n id of C: "; C.display();
    cout<<"\n id of D: "; D.display();
}
```

// program //

ecopycon.cpp

// initialize objects using default copy constructor

```
#include <iostream>
using namespace std;
class  Distance    //English Distance class
{
private:
int feet;
float inches;
public:
//constructor (no args)
Distance() : feet(0), inches(0.0)
{ }
//Note: no one-arg constructor
//constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << '\"'; }
```

```
};
int main()
{
Distance dist1(11, 6.25); //two-arg constructor
Distance dist2(dist1); //one-arg constructor
Distance dist3 = dist1; //also one-arg constructor
//display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

ecopycon.cpp

```
// initialize objects using default copy constructor
#include <iostream>
using namespace std;
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
//constructor (no args)
Distance() : feet(0), inches(0.0)
{ }                                          //Note: no one-arg constructor
Distance(int ft, float in) : feet(ft), inches(in)       //constructor (two args)
{ }
```

```
void getdist()                          //get length from user
{
cout << "\nEnter feet: "; cin >> feet;   cout << "Enter inches: "; cin >>
inches; }
void showdist()                         //display distance
{ cout << feet << "\'-" << inches << '\"'; }
};
void  main()
{
Distance dist1(11, 6.25);               //two-arg constructor
Distance dist2(dist1);                  //one-arg constructor
Distance dist3 = dist1;                 //also one-arg constructor
//display all lengths
cout << "\ndist1 = "; dist1.showdist();   cout << "\ndist2 = ";
dist2.showdist();
cout << "\ndist3 = "; dist3.showdist(); cout << endl; }
```

دالة البناء:- عند تشغيل البرنامج من غير أي استدعاء هي عبارة عن دالة تنفذ تلقائيا

أو بصورة أخرى-:

هي عبارة عن دالة يتم استدعاؤها مباشرة عند اشتقاق كائن من صنف معين

دالة الهدم:- هي عبارة عن دالة يتم استدعاؤها مباشرة عند اشتقاق كائن من صنف ولكن عند نهاية البرنامج.

مثال توضيحي لدوال البناء والهدم-:

```
#include<iostream>
#include<stdlib.h>
using namespace std;
class x
{
public:
```

```
x( );// بناء دالة
~x( );// هدم دالة
};
x::x( )
{
cout<<"constructor is called \n";
}
x::~x( )
{
cout<<"destructor is called \n";
system("PAUSE" );
}
int main( )
{
x x1,x2;
return 0;}
```

1- يتم استدعاء دالة البناء عند اشتقاق الكائن وبصورة تلقائية
2- دالة الهدم يتم استدعاؤها أيضا بصورة تلقائية ولكن عند نهاية البرنامج
3- كل كائن يستدعي دالة البناء ودالة الهدم

خواص دوال البناء والهدم-:

1- تحمل نفس اسم الفئة ولكن دالة الهدم تسبق بعلامة **(~)**
2- يتم تعريفهم في مستوى الحماية العام **public**
3- يمكن إنشاء أكثر من دالة بناء
4- يمكن إنشاء دالة هدم واحدة فقط
5- ليس لهما أنواع رجوع

مثال:- برنامج يقوم بإيجاد العمليات الأساسية(الجمع, الطرح , الضرب , القسمة) لعددين مدخلين من قبل المستخدم باستخدام دالة البناء.

```
#include<iostream>
using namespace std;
class operations
{
float a,b,c; int
public:
operations( );
void result( );
};
operations::operations( )
{
cout<<"Mathematical Operations \n";
cout<<" 1- Addition \n";
cout<<" 2- Subtraction \n";
cout<<" 3- Multiplication \n";
cout<<" 4- Division \n";
cout<<" Please Enter Tow Values a and b \n";
cin>>a>>b;
}
void operations::result( )
{
c=a+b;
cout<<a<<"+"<<b<<"="<<c<<"\n";
c=a-b;
cout<<a<<"-"<<b<<"="<<c<<"\n";
c=a*b;
cout<<a<<"*"<<b<<"="<<c<<"\n";
```

```
if(b!=0)
{
c=a/b;
cout<<a<<"/"<<b<<"="<<c<<"\n";
}
else
cout<<"the result of division is infinite \n";
}
int main( )
{
operations op;
op.result( );
return 0;
}
```

دالة البناء التي تستخدم المعاملات:- يمكن أن نستخدم أي نوع من المعاملات **parameters** في داخل دالة البناء وذلك من خلال إضافة عدد المعاملات الضرورية في دالة البناء المعلنة

مثال-:

```
#include<iostream>
using namespace std;
class myclass
{
int a,b;
public:
myclass(int,int);
void show( );
};
myclass::myclass(int x,int y)
```

```
{
cout<<"constructor function \n";
a=x;
b=y;
}
void myclass::show( )
{
cout<<a<<"\t"<<b<<"\n";
}
int main( )
{
myclass ob(4,7);
ob.show( );
return 0;
}
```