Constructor

We have observed in our previous topics that, to initialize the data members of a class object we call a dedicated member function of that class.

Example:

```
#include<iostream.h>
#include<conio.h>
class x
{
        int a;
        int b;
public:
        void input(int, int);
        void display();
};
void x::input(int x, int y)
{
        a=x;
        b=y;
}
void x::display()
{
        cout<<a<<"\n";
        cout<<b<<"\n";
}
void main()
{
        clrscr();
        x x1,x2;
        x1.input(10,20);
        cout<<"\n";
        x2.input(30,40);
        x1.display();
        x2.display();
getch();
}
```

When the object of class ‘x’ is created the using the statement
x x1;
the memory is allocated to for the private members of the object of class ‘x’, but those data members are not initialized. To initialize the data members a specific function like input is dedicated with reference to that particular object.

Now, the question is, can’t we initialize members of class automatically, as soon as its object is created. It is possible and is done by specific function in C++ known as constructor.

We stated earlier that one of the aims of C++ is to create user-defined, data types such as class, that behave very similar to the built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

int x=10;
float x=3.5;

Are valid initialization statements for basic data types. Similarly, when a variable of built in type get out of scope the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. 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 object. It also provides another member function called the destructor that destroys the objects when they are no longer required.

Declaration and definition of constructor

A **constructor** (having the same name as that of the class) is a member function which is automatically used to initialize the objects of the class type with legal initial values. As the name of constructor is same as its class name, it is called as special member function, with only restriction that it cannot return any value.

Example:

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
        int b;
public:
        A();  //constructor Declaration without parameter
        void display();
};
A::A() // constructor Definition outside the class
{
        a=10;
        b=20;
}
void A::display()
{
        cout<<a<<"\n";
        cout<<b<<"\n";

}
void main()
{
        clrscr();
        A a1; //constructor will be invoked with creation of object
        cout<<"constructor invoked for object a1";
        cout<<"\n";
        a1.display();
        A a2;
        cout<<"constructor invoked for object a2";
        cout<<"\n";
        a2.display();
```

```
getch();
}
```

The constructor A( ) is automatically called and the data members are initialized. There is no need to call the constructor explicitly using an object for initializing the data members.

What happens when constructor is not defined or declared explicitly, but object is created
Consider an example:

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
public:
        void input(int);
        void display( );
};
void A::input(int x)
{
        a=x;
}
void A::display( )
{
        cout<<a<<"\n";
}
void main()
{
        clrscr( );
        A a1,a2;
        a1.input(10);
        cout<<"\n";
        a2.input(30);
        a1.display( );
        a2.display( );
getch( );
}
```

When the program is compiled, the compiler itself embeds a call to a constructor implicitly which does nothing. It means, it implicitly calls a constructor called “default constructor” which has empty definition. During compilation process, the above program becomes like as shown below:

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
public:
        void input(int);
```

```
        void display( );
        A( );    //prototyped implicitly by the compiler
};
A::A( )  //constructor implicitly defined by the compiler
{
        //empty definition
}
void A::input(int x)
{
        a=x;
}
void A::display( )
{
        cout<<a<<"\n";
}
void main()
{
        clrscr( );
        A a1;
        a1.input(10);
        cout<<"\n";
        a1.display( );
getch( );
}
```

When object is created as,

A a1;

During compilation the above statement is transformed into,

a1.A( );

i.e., the constructor is called implicitly by the compiler as shown above, which is then transformed into,

A(&a1);

Special Characteristics of Constructors

1. These have some special characteristics. These are given below:
2. These are called automatically when the objects are created.
3. All objects of the class having a constructor are initialized before some use.
4. These should be declared in the public section for availability to all the functions.
5. Return type (not even void) cannot be specified for constructors. These cannot be inherited, but a derived class can call the base class constructor.
6. These cannot be static.
7. Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.
8. These can have default arguments as other C++ functions.

9. A constructor can call member functions of its class.
10. An object of a class with a constructor cannot be used as a member of a union.
11. A constructor can call member functions of its class.
12. We can use a constructor to create new objects of its class type by using the syntax.
13. The make implicit calls to the memory allocation and deallocation operators new and delete.
14. These cannot be virtual.

Types of constructor

Default constructor

A constructor that has zero parameter list or a constructor that accepts no argument is called as a zero argument constructor or default constructor. If such constructor is not defined then, the compiler defines the default constructor implicitly during compilation. If the default constructor is defined explicitly then, compiler will not define the constructor implicitly, but it calls the constructor implicitly.

Parameterized Constructors

The constructors that can take arguments are called parameterized constructor.

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
        int b;
public:
        A(int x, int y)
        {
                a=x;
                b=y;
        }
        void display();
};
void A::display()
{
        cout<<a<<"\n";
        cout<<b<<"\n";
}
void main()
{
        clrscr();
        A a1(10,20);
        cout<<"constructor invoked for object a1";
        cout<<"\n";
        a1.display();
        A a2(30,40);
        cout<<"constructor invoked for object a2";
        cout<<"\n";
        a2.display();
```

```
getch();
}
```

When a constructor has been parameterized, the abject declaration statement such as
A a1;
may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

1. By calling the constructor explicitly.
2. By failing the constructor implicitly.

The following declaration illustrates the first method:

A a1=A(10,20) // explicit call

This statement creates an object ‘a1’ and passes the values 10 and 20 to it, The second is implemented as Follows:

A a1(30,40); // implicit call

This method, sometimes called the shorthand method, in used very often as it is shorter, looks better and is easy to implement. Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor.

One important thing to be noted is that, when parameterized constructor is define and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence, creating a simple objects as

A a3;

Will flash an error.

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
        int b;
public:
        A(int x, int y)
        {
                a=x;
                b=y;
        }
        void display();
};
void A::display()
{
        cout<<a<<"\n";
        cout<<b<<"\n";
}
```

```
void main()
{
        clrscr();
        A a1(10,20);
        cout<<"constructor invoked for object a1";
        cout<<"\n";
        a1.display();
        A a2(30,40);
        cout<<"constructor invoked for object a2";
        cout<<"\n";
        a2.display();
        A a3;   //Flashes error
getch();
}
```

To resolve this we should explicitly define a default constructor as follows:

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a;
        int b;
public:
        A( ) {}
        A(int x, int y)
        {
                a=x;
                b=y;
        }
        void display();
};
void A::display()
{
        cout<<a<<"\n";
        cout<<b<<"\n";
}
void main()
{
        clrscr();
        A a1(10,20);
        cout<<"constructor invoked for object a1";
        cout<<"\n";
        a1.display();
        A a2(30,40);
        cout<<"constructor invoked for object a2";
        cout<<"\n";
        a2.display();
        A a3;
getch();
}
```

Multiple Constructors in a Class
So far we have used two kinds of constructors. They are:

A( ) ; ***// No arguments***
A (int x, int y); ***// Two arguments***

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

```
class integer
{
        int m. n;
public:
        integer( )
        {
                m=0; n=0
        }
        Integer(int a, int b)
        {
                m=a;
                n=b;
        }
        integer (integer & i)
        {
                m=i.m;
                n=i.n;
        }
};
```

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

integer i;

would automatically invoke the first constructor and set both m and n of B to zero. The statement

integer i2(20,40)

would call the second constructor which will initialize the data members m and n of i2 to 20 and 40 respectively, Finally, the statement

integer i3 (i2);

would invoke the third constructor which copies the values of i2 into i3. In other words, it sets the value of every data element of i3 to the value of the corresponding data element of i2. such a constructor is called the copy constructor.

Overloaded constructor

The process of sharing the same name by two or more Functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Example:

```
#include<iostream.h>
#include<conio.h>
class complex
{
        float real;
        float img;
public:
        complex()
        {

        }
        complex (float a)
        {
                real=img=a;
        }
        complex( float r, float i)
        {
                real=r;
                img=i;
        }
        friend complex sum(complex , complex );
        void show(complex);

};
complex sum(complex c1, complex c2)
{
        complex c3;
        c3.real=c1.real + c2.real;
        c3.img=c1.img+c2.img;
        return (c3);
}
void complex :: show( complex c)
{
        cout<<c.real<<"+j"<<c.img<<"\n";
}
void main()
{
        clrscr();
        complex c1(10,20);
        complex c2(20.5);
        complex c3;
        c3=sum(c1,c2);
        cout<<"first number:"<<"\n";
        c1.show(c1);
```

```
        cout<<"second number:"<<"\n";
        c2.show(c2);
        cout<<"sum of two numbers:";
        c3.show(c3);
getch();
}
```

Let us look at the first constructor again.
complex();

It contains the empty body and does not do anything. This is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now? C++ compiler has an implicit constructor which creates objects, even though it was not define in the class.
This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

Copy constructor
A copy constructor is used to declare and initialize an object from another object. 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 i2 i1;
The process of initializing through a copy constructor is known as copy initialization. Remember, the statement

*A copy constructor takes a reference to an object of the same class as itself as an argument.
Example:

```
#include<iostream.h>
#include<conio.h>
class a
{
        int p;
        int b;
public:
        a (a& x)
        {
                p=x.p;
                b=x.b;
        }
        a(int x, int y)
        {
                p=x;
                b=y;
        }
        void display()
        {
                cout<<p<<"\n"<<b<<"\n";
```

```
        }

};
void main()
{
        clrscr();
        a a1(10,20);
        a a2(30,40);
        cout<<"1st object called"<<"\n";
        a1.display();
        cout<<"2nd object called"<<"\n";
        a2.display();
        a a3=a2;
        cout<<"values from copy constructor"<<"\n";
        a3.display();
getch();
}
```

When are copy constructor called?

Whether the program defines the copy constructor or compiler does t implicitly, in either case, the copy constructor is called in the following three situations.

1. The initialization of one class object by another object of the same class by equating the newly created object to the existing object.

```
#include<iostream.h>
Class code
{
        int id;
public:
        code(int a)
        {
                Id=a;
        }
        Void display()
        {
                Cout<<id<<”\n”;
        }
};

int main()
{
        Code obj1(10);
        Obj1.display();
        Code  obj2(obj1);   //or  code obj2=obj1;
        Obj2.display();
return 0;

}
```

In the above example, the copy constructor is called for object obj2 and object obj1 is passed as its reference parameter.

2. When an object is passed as an non-reference to a function.

```
#include<iostream.h>

class cal
{
        public:
                int a,b;
                int add(cal d)
                {
                        Return(d.a+d.b);
                }
};
int main()
{
        Cal c;
        c.a=10;
        c.b=20;
        int ret=c.add(c);
        cout<<ret<<"\n";
}
```

3. When the function returns the class object as its return value. The return object is equated to the newly creating object.

```
 Class A
{
        public:
                A()
                {

                }
                A(int x)
                {
                        A=x;
                }
                A abc();
};
A A::abc()
{
        A a3;
        a3.a=a;
        return a3;
}
int main()
{
        A a1(200);
        A a2=a1.abc();
```

```
        cout<<a2.a<<"\n";
return(0);
}
```

Constructor with default arguments

It is possible to define constructors with default arguments. For example, the constructor 'complex ( )' can be declared as follows:

complex (float real, float imag=0);

The default value of the argument imag is zero. Then, the statement

complex C(5,0);

assigns the value 5.0 to the real variable and 0.0 to imag (by default). However, the statement

complex C(2.0,3.0);

assigns 2.0 to real and 3,0 to imag. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor A::A( ) and the default argument constructor A::A(int =0). The default argument constructor can he called with either/ one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as
A a
The ambiguity is whether to call A::A( ) or A::A(int=0)

Parameterized Dynamic Constructor

Constructor that allocate memory dynamically for an object at the time of their construction are known as parametrized dynamic constructor.