Constructor

By Shailee Shah
Assistant professor
President Institute of Computer Application

Constructors

- In all the cases, we have used member functions such as *putdata()* and setvalue() to provide initial values to the private member variables.
- Example,

```
A.input();
X. getdata(10,20);
```

- All these “function call” statement are used with the appropriate objects that have already been created.
- These function can’t be used to initialize the member variables at the time of creation of their objects.
- 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.
- Example,

```
int m = 20; // is valid for basic data types.
```

- It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer required.

- C++ provide 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.
- A constructor is *'special'* member function whose task is to initialize the objects of its class.
- 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 constructs the values of data member of the class.

```
class ABC{
                    int m, n;
          public :
                    ABC();                  // constructor declared
                    . . . .
                    . . . .
};
ABC :: ABC(){                               // constructor defined
          m = 0;
          n = 0;
}
```

- When a class contains a constructor like the one defined above, it is guaranteed that object created by the class will be initialized automatically.
- Example,

  ABC a1;
- This declaration will initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor function. Like, a.putdata();
- There is a problem if there are a large number of objects.
- A constructor that accepts no parameters is called the default constructor.
- The default constructor for class ABC is **ABC :: ABC()**
- If no such constructor is defined, then the compiler supplies a default constructor.
- ABC a1; invokes the default constructor of the compiler to create the object a1;

Characteristics of Constructors

- 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, and they can't return values.
- They can't be inherited, though a derived class can call the base class constructor.
- Like other C++ function, they can have default arguments.
- Constructors can not be **virtual**.
- We can not refer to their address.
- An object with a constructor (or destructor) can not be used as a member of a union.
- They make 'implicit call' to the operators new and delete when memory allocation is required.

Constructors

```
#include<iostream.h>
#include<conio.h>
class ABC
{
                int x,y;
        public:
                ABC()
                {
                        x=10;
                        y=20;
                }
                void show()
                {
                        cout<<"X = " << x;
                        cout<<"Y = " << y;
                }
};
```

```
void main(){
        clrscr();
        ABC a1;
        a1.show();
        getch();
}
```

Output:

X=10
Y=20

Parameterized Constructor

- C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.
- The constructors that can take argument are called *parameterized constructors*.
- The constructor ABC() may be modified to take arguments as shown below :

```
Class ABC
{
        int m, n;
        public:
                ABC(int, int);
};
ABC : ABC(int x, int y)
{
        m = x;
        n = y;
}
```

- When a constructor has been parameterized, the object declaration statement such as,

  ABC a1; may not work.

- We must pass the initial values as arguments to the constructor function when an objects is declared.
- This can be done in two ways :

1> By calling the constructor explicitly :

```
ABC a1 = ABC(100, 200);
```

This statement creates an **ABC** object **a1** and passes the values 100 and 200 to it.

2> By calling the constructor implicitly :

```
ABC a1(100, 200);
```

- This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.
- **Note :** When the constructor is parameterized, we must provide appropriate arguments for the constructor.

```
#include<iostream.h>
#include<conio.h>
class ABC
{
        int m, n;
    public:
        ABC(int x, int y);
        void show()
        {
            cout<<"M = " << m <<"\n";
            cout<<"N = " << n <<"\n";
        }
};
ABC :: ABC(int x, int y)
{
    cout<<“INSIDE PARAMETERIZED CONSTRUCTOR”<<endl;
    m = x;
    n = y;
}
```

```
void main()
{
        clrscr();
        ABC a1(100, 200);
        ABC a2 = ABC(300, 400);
        a1.show();
        a2.show();
        getch();
}
```

Output:

INSIDE PARAMETERIZED CONSTRUCTOR
INSIDE PARAMETERIZED CONSTRUCTOR
M=100
N=200
M=300
N=400

```
#include<iostream.h>
#include<conio.h>
class ABC
{
        int m, n;
    public:

    ABC()    //default constructor
    {
        cout<<“DEFAULT CONSTRUCTOR”<<endl;
    }
    ABC(int x, int y)
    {
            cout<<“INSIDE PARAMETERIZED
            CONSTRUCTOR”<<endl;

            m = x;
            n = y;
    }
    void show()
    {
            cout<<"M = " << m <<"\n";
            cout<<"N = " << n <<"\n";
    }
};
```

```
void main()
{
        clrscr();
        ABC x;
        ABC a1(100, 200);
        a1.show();

        getch();
}
```

Output:

DEFAULT CONSTRUCTOR
INSIDE PARAMETERIZED
CONSTRUCTOR
M=100
N=200

- The parameters of constructor can be of any type except that of the class to which it belongs.

  ABC :: ABC(ABC); // is invalid

- However, a constructor can accept a reference to its own class as a parameter (*copy constructor*).

  ABC :: ABC(ABC&); // is valid (copy constructor)

Constructor Overloading

- We have used two kinds of constructor :

  ABC();

  ABC(int, float);
- 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 constructors in the same class.
- Example,
- hen more than one constructor function is defined in a class, we says that the **constructor is overloaded.**

```
class ABC{
        int m,n;
    public:
        ABC(){m=0; n=0;}
        ABC(int a, int b){m = a; n = b;}
        ABC(int x){ m = n = x; }
};
```

Constructor Overloading

```
#include<iostream.h>
#include<conio.h>
class ABC{
        int m, n;
    public:
        ABC()       //DEFAULT
        {
            m=0;
             n=0;
        }
        ABC(int a, int b) //PARAMETERIZED 2
        {
            m = a;
            n = b;
        }
        ABC(int x) //PARAMETERIZED 1
        {
            m = n = x;
        }
        void show()
        {
            cout<<"M = " << m <<"\n";
            cout<<"N = " << n <<"\n";
        }
};
```

```
void main(){
        clrscr();
        ABC a1;
        ABC a2(100, 200);
        ABC a3 = ABC(300);
        a1.show();
        a2.show();
        a3.show();
        getch();
}
```

Output:

M=0
N=0

M=100
N=200

M=300
N=300

Constructor Overloading

```
ABC(){ }
```

- It contains the empty body and does not do anything.
- We just stated that this used to create objects without any initial values.
- 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 defined in the class.
- This works fine as long as we do not use any other constructor 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.

Constructor Overloading

```
#include<iostream.h>
#include<conio.h>
class rect{
        int height,width;
    public:
        rect()
            {
                height=0;
                width=0;
            }
        rect(int h,int w)
            {
                height=h;
                width=w;
            }
        void show()
        {
            cout<<“Height is  = " << height <<"\n";
            cout<<“Width is  = " << width <<"\n";
        }
};
```

```
void main(){
        clrscr();
        rect r1;
        r1.show();

        rect r2(5,10);
        r2.show();

        getch();
}
```

Output:

Height is=0
Weight is=0

Height is=5
Weight is=10

Constructor with Default Argument

- It is possible to define constructors with default arguments.
- 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 function calls are,

  complex c1(5.2);

  complex c2(5.2, 2.3);

- 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 be 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).**

Constructor with Default Argument

```
#include<iostream.h>
#include<conio.h>
class ABC
{
        int m,n;
    public:
        ABC ()
            {
                cout<<“INSIDE DEFAULT”<<endl;
                m=40;
                n=20;
            }
        ABC (int a,int b=90)
            {
             cout<<“parameterized DEFAULT
ARGUMENT”<<endl;
                m=a;
                n=b;
            }
void show()
        {
            cout<<“M =" << m<<"\n";
            cout<<“N  = " << n<<"\n";
        }
};
```

```
void main(){
        clrscr();
        ABC a1;
        ABC a2(10);
        a1.show();
        a2.show();

        getch();
}
```

Output:

INSIDE DEFAULT
M=40
N=20

parameterized DEFAULT
ARGUMENT
M=10
N=60

Copy Constructor

- A copy constructor is used to declare and initialize an object from another object.

  ABC a2 (a1);

- It would define the object a2 and at the same time initialize it to the values of a1.
- Another form of this statement is

  ABC a2 = a1;

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

Note :-

a2 = a1; is a **valid** but will not invoke the copy constructor.

- if a1 and a2 are objects, this statement is legal and simply assigns the values of a1 to a2, member-by-member. (task of overloaded assignment operator)
- A copy constructor takes a reference to an object of the same class as itself as an argument.

Copy Constructor

```
#include<iostream.h>
#include<conio.h>
class CODE
{
        int id;
   public:
        CODE()
        {

        }
        CODE(int a)
        {
                id = a;
        }
        CODE(CODE & x)
        {
                id = x.id;
        }
        void show()
        {
                cout<<"ID = " << id;
        }
};
```

```
void main(){
   clrscr();
   CODE A(100);
   CODE B(A); / /copy

   CODE C = A; //copy
   CODE D;
   D = A;
   cout<<"\n id of A = ";
   A.show();
   cout<<"\n id of B = ";
   B.show();
   cout<<"\n id of C = ";
   C.show();
   cout<<"\n id of D = ";
   D.show();
   getch();
}
```

```
Class demo
{
   int p,q;
   public:
            demo()
            {
                     p=2;
                     q=4;
            }
            demo(int x,int y)
            {
                     p=x;
                     q=y;
            }
            demo(demo & a)
            {
                     p=a.p;
                     q=a.q;
            }
            void show()
            {
                     cout<<"P= "<<p<<endl;
                     cout<<"Q= "<<q<<endl;
            }
};
```

```
void main()
{
   clrscr();
   demo d1;
   cout<<"1st object details"<<endl;
   d1.show();

   demo d2(20,40);
   cout<<"2nd object details"<<endl;
   d2.show();

   demo d3(d2);
   cout<<"3rd object details"<<endl;
   d3.show();

   demo d4=d2;
   cout<<"4th object details"<<endl;
   d4.show();
   getch();

}
```

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.

```
~ABC() {  }
```

- 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(of block of function as the case may be) 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.

Destructor

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

int count = 0;
class ABC
{
   public:

      ABC()
      {
         count++;
         cout<<"\nNo. of Object created : "<<count;
       }

      ~ABC()
      {
         cout<<"\nNO. of Object destroyed : "<<count;
         count--;
      }
};
```

```
void main()
{
   clrscr();
   cout<<"\n\nEnter Main\n";
      {
            ABC a1,a2,a3,a4;
            {
            cout<<"\n\nBlock1";
            ABC a5;
            }

            {
            cout<<"\n\nBlock2";
            ABC a6;
            }

        cout<<"\n\nBlock Main\n";

       }
   getch();
}
```

Output:

```
Enter main
No. of object created 1
No. of object created 2
No. of object created 3
No. of object created 4

Enter Block1
No. of object created 5
No. of object  destroyed 5

Enter Block2
No. of object created 5
No. of object  destroyed 5

Re Enter main
No. of object destroyed 4
No. of object destroyed 3
No. of object destroyed 2
No. of object destroyed 1
```

Thank you