Object Oriented Programming Using Cpp

Topic:

Constructors And Destructors

Chapter 7 Constructors and Destructors

7.1 Introduction

We know that if a variable is declared but not initialized then it contains some garbage value. The compiler is not capable of doing the initialization itself. The programmer needs to perform the initialization.

Similarly, in C++, when an object is declared then its data members contain garbage value if no initialization is done. In previous chapter, we have seen that a member function can be used for the initialization purpose. However, we need to call that member function whenever we need to do the initialization. Now, this is something difficult for the user to call the member function again and again. So, the requirement is to find any way in which initialization can be done automatically i.e. User need no to call the member function separately. One way is the use of static data member where if user does not provide any value, system provides default value to the data member and initialization is done. However, there is only static data member possible in a class. So, again the problem is same for rest data members. Next solution was provided using static object where all the data members of an object is initialized with default value. But again the restriction was that only one object can be made static. So, at present, user faces two requirements as mentioned below in terms of initialization of data members:

- All the data members of an object should be initialized automatically
- There must not be any restriction on number of objects to be created in the program.

Now, C++ provides a mechanism, known as Constructors, to provide an efficient solution for the above mentioned requirements.

7.2 Constructors

Constructor is a special member function which automatically initializes all the data members of an object at the moment the object is created.

Properties of constructors

- Constructor shares the class name. i.e. A constructor is always defined with the class name.
- A constructor does not have any return type, not even void. The reason behind this is, the main purpose of constructor is the initialization of data members. So, performing any operation must be avoided inside a constructor.

- Constructor is executed automatically whenever an object is created. So, it initializes all the data members for that object as per the definition of the constructor. However, a constructor can also be called explicitly.
- Like functions, constructors may or may not accept arguments. A constructor without any argument is known as default constructor.
- Constructor may be defined as private or public. However, the general practice is to define constructor as public.
- Constructor can be defined inside the class as well as outside the class using the same syntax what had been used for defining member functions outside the class.
- Constructors can be overloaded. i.e. A class may contain more that one constructor. Now, again the user needs to follow the rules of function overloading to avoid ambiguity for the system.
- Constructors can't be inherited.
- Constructors can't be virtual.

7.2.2 Types of constructors

Broadly constructors are divided into following types:

1. Default constructor
2. Parameterized constructor
3. Constructor with default argument
4. Copy constructor

7.2.2.1 Default constructor

Constructor without any argument is known as default constructor. This constructor is called automatically whenever an object is declared for that class. The moment object is created, constructor is called and it initializes all the data members for the object as per the constructor definition.

Syntax:

```
class <class_name>
{


        public:
        <class_name>()        // Default constructor
        {
        }
    };
```

Note: If user wishes to define the constructor outside the class, then they must give the prototype inside the class and then they can define the constructor outside the class using class name and scope resolution operator.

Program 36 Write a C++ program to illustrate the concept of default constructor.

Code:

```
    #include<iostream.h>
    #include<conio.h>
    class  NUMBER
    {
        int     n1,n2,n3;
        public:
            NUMBER()            // Default constructor
            {
                n1=5;
                n2=10;
                n3=15;
            }
            void   display()
            {
                cout<<"n1="<<n1<<endl;
                cout<<"n2="<<n2<<endl;
                cout<<"n3="<<n3<<endl;
            }
    };
```

```
void   main()
{
        NUMBER    ob1;      // constructor is called 1st time
        ob1.display();
        NUMBER    ob2;      //.constructor is called 2nd time
        ob2.display();
        NUMBER    ob3;      //constructor is called 3rd time
        ob3.display();
}
```

O/P:

```
n1=5          //for object ob1
n2=10
n3=15
n1=5          //for object ob2
n2=10
n3=15
n1=5          //for object ob3
n2=10
n3=15
```

Note: In this example, we have used compile time method in the constructor definition due to which the data members of all the three objects has the same value. This can be avoided by using runtime initialization method while defining the constructor. This can be done as below:

```
NUMBER()
{
        cout<<"Enter the values for n1, n2 and n3"<<endl;
        cin>>n1>>n2>>n3;
}
```

O/P:

```
Enter the values for n1,n2 and n3                // for ob1
2      3      4
n1=2
n2=3
n3=4
Enter the values for n1,n2 and n3                // for ob2
10     20     30
n1=10
n2=20
n3=30
Enter the values for n1,n2 and n3                // for ob3
11     22     33
n1=11
n2=22
n3=33
```

7.2.2.2 Parameterized constructor

Constructor which accepts argument is known as parameterized constructor. This constructor is called automatically whenever an object is declared for that class. However, this time arguments must be passed while declaring the object. The moment object is created, constructor is called and it initializes all the data members for the object as per the constructor definition.

Syntax:

```
class <class_name>
{
        public:
```

```
        <class_name>(argument)        // Parameterized constructor
        {


        }
    };
```

Note: If user wishes to define the constructor outside the class, then they must give the prototype inside the class and then they can define the constructor outside the class using class name and scope resolution operator.

Program 37 Write a C++ program to illustrate the concept of parameterized constructor.

Code:

```
    #include<iostream.h>
    #include<conio.h>
    class  NUMBER
    {
        int     n1,n2,n3;
        public:
        NUMBER(int   x, int   y, int   z )   // Parameterized constructor
        {
            n1=x;
            n2=y;
            n3=z;
        }
        void   display()
        {
            cout<<"n1="<<n1<<endl;
            cout<<"n2="<<n2<<endl;
            cout<<"n3="<<n3<<endl;
        }
    };
    void   main()
    {
            NUMBER     ob1(5,10,15);
            ob1.display();
            NUMBER     ob2(2,3,4);
            ob2.display();
    }
```

O/P:

```
    n1=5          //for object ob1
    n2=10
    n3=15
    n1=2          //for object ob2
    n2=3
    n3=4
```

Now, to implement runtime initialization in case of parameterized constructor, the main function will be as below:

```
    void  main()
    {
            int     a,b,c;
            cout<<"Enter a,b and c";
            cin>>a>>b>>c;
            NUMBER     ob1(a,b,c);
            Ob1.display();
    }
```

Note:

Constructor overloading: A class may contain more than one constructor. The concept is known as constructor overloading. However, while overloading constructors, user must take care of making the constructors different from each other so that system can call the constructor successfully. For this, the same three rules for function overloading are applicable. Again, while declaring object, user needs to take care as constructor execution depends on the type of object created.

Program 38 Write a C++ program to illustrate the concept of constructor overloading.

Code:

```
#include<iostream.h>

#include<conio.h>

class  NUMBER

{

        int       n1,n2,n3;

        public:

        NUMBER()                          // Default constructor

        {

                cout<<"Enter the values for n1, n2 and n3"<<endl;

                cin>>n1>>n2>>n3;

        }

        NUMBER(int    x, int   y, int   z )   // Parameterized constructor

        {

                n1=x;

                n2=y;

                n3=z;

        }
```

```
          void   display()
          {
                    cout<<"n1="<<n1<<endl;
                    cout<<"n2="<<n2<<endl;
                    cout<<"n3="<<n3<<endl;
          }
};
void   main()
{
          NUMBER    ob1;       // Default constructor is called
          ob1.display();
          NUMBER    ob2(2,3,4);    //.Parameterized constructor is called
          ob2.display();
}
```

O/P:

```
Enter the values for n1, n2 and n3
10    20    30
n1=10       // Default constructor
n2=20
n3=30
n1=2        // Parametrized constructor
n2=3
n3=4
```

7.2.2.3 Constructor with default argument

It is a special case of parameterized constructor where each argument is assigned with a default value. So, if user passes all arguments during object declaration, system uses those values. But if user passes less number of arguments during object declaration, system uses default values for missing parameters.

Program 39 Write a C++ program to illustrate the concept of constructor with default argument.

Code:

```
#include<iostream.h>
#include<conio.h>
class  NUMBER
{
        int     n1,n2,n3;
        public:
        NUMBER(int   x=2, int  y=4, int   z=6)
        {
                n1=x;
                n2=y;
                n3=z;
        }
        void   display()
        {
                cout<<"n1="<<n1<<endl;
                cout<<"n2="<<n2<<endl;
                cout<<"n3="<<n3<<endl;
        }
};
void   main()
{
        NUMBER     ob1(5,10,15);
        ob1.display();
        NUMBER     ob2(11,12);
        ob2.display();
        NUMBER ob3(20);
        ob3.display();
        NUMBER ob4;
        ob4.display();
    }
```

O/P:

```
    n1=5        //for object ob1
    n2=10
    n3=15
    n1=11       //for object ob2
    n2=12
    n3=6
    n1=20       //for object ob3
    n2=4
    n3=6
    n1=2        //for object ob4
    n2=4
    n3=6
```

Explanation: Here, in 1st object declaration, user passes all the three arguments. So, system accepts these values and output comes as 5, 10 and 15. In 2nd object declaration, user passes two arguments. So, system uses default value for 3rd parameter and the output comes as 11, 12 and 6. In 3rd object declaration, user passes only one argument. So, system takes default values for rest two arguments and output comes as 20, 4 and 6. Now in 4th object declaration, user does not pass any argument. So, if class contains default constructor then that constructor will be executed. But here, the class does not contain any default constructor, so system executes constructor with default argument and it uses default argument for all 3 parameters and output is 2, 4 and 6. However, this is a contradiction as in 4th case the object declaration is not supporting any parameters.

7.2.2.4 Copy Constructor

A constructor which accepts reference of an object as its argument is known as copy constructor.

Syntax:

```
class <class_name>

{

        ......................;

        public:

        <class_name>(class_name&obj)

        {


        }

};
```

Program 40: Write a C++ program to illustrate the concept of copy constructor.

Code:

```
#include<iostream.h>

#include<conio.h>

class NUM

{

        int     n1;

        public:

        NUM()       //Default argument

        {

                n1=0;

        }
        NUM(int  k)           //Parameterized constructor
        {
              n1=k;
        }
        NUM(NUM  &j)          //Copy constructor
        {
              n1=j.n1;
        }
        void  show()
        {
              cout<<"n1="<<n1<<endl;
        }
};
void main()
{
        NUM  ob1(50);         //Parametrized constructor is called
        ob1.display();
        NUM  ob2(ob1);        //Implicit Copy constructor is called
        ob2.display();
        NUM  ob4;             //Default constructor is called
        getch();
}
```

O/P:

```
n1=50
n1=50
n1=50
```

7.3 Calling constructors explicitly

Till now we have seen that a constructor is called implicitly whenever an object is declared. It is also possible to call a constructor explicitly. However, it is generally avoided to prevent any problem.

Now, a constructor can be called explicitly in following two ways:

- Like a normal function
- Using class and its object along with scope resolution operator.

Program 41: Write a C++ program to illustrate the concept of explicit call to a constructor.

Code:

```
#include<iostream.h>
#include<conio.h>
class  SAMPLE
{
        int     n1;
        public:
                SAMPLE()
                {
                        n1=5;
                        cout<<"n1="<<n1<<endl;
                }
                SAMPLE(int     x)
                {
                        n1=x;
                        cout<<"n1="<<n1<<endl;
                }
};
```

```
void main()
{
        SAMPLE();               // Explicit call (like a function) to constructor
        SAMPLE(10);             // Explicit call (like a function) to constructor
        SAMPLE  s1;             //Implicit call to constructor
        s1.SAMPLE: :SAMPLE();   //Explicit call to constructor
        s1.SAMPLE: :SAMPLE(20); //Explicit call to constructor
        getch();
}
```

O/P:

```
n1=5        //Due to explicit call to default
n1=10       //Due to explicit call to parametrized
n1=5        //Due to implicit call as object is declared
n1=5        //Due to explicit call to default
n1=20       //Due to explicit call to parametrized
```

7.4 Destructors

Destructor is also a special member function which destroys the object once its requirement is complete. It is done to release the memory space occupied by the object so that the released memory can be used in some other purpose.

Features of destructor

- Destructor also has the class name. To make a distinction between destructors and constructors, **tilde (~)** symbol is used with class name in case of destructors.
- Destructor also does not have any return type.
- Like constructors, destructors also can't be inherited.
- Destructors can't be overloaded i.e. A class can have only one destructor.
- Destructor is executed implicitly just before completion of the program. However, it is also possible to call a destructor explicitly but it is generally avoided.

Program 42: Write a C++ program to illustrate the concept of destructors.

Code:

```
#include<iostream.h>
#include<conio.h>
class  SAMPLE
{
        int     a;
        public:
        SAMPLE()
        {
                cout<<"Enter a";
                cin>>a;
                cout<<"a="<<a<<endl;
        }
        ~SAMPLE()
        {
                cout<<"Destructor activates";
        }
};
void main()
{
        SAMPLE    s1;
        SAMPLE    s2;
        getch();
}
```

O/P:

```
Enter a              //Constructor is called
```

```
10

a=10

Enter a                 //Constructor is called

20

a=20

Destructor activates         //Destructor is called

Destructor activates         //Destructor is called
```

Note:

1. Like constructors, destructors can also be defined outside the class using the same syntax what we use to define a member function outside the class.

2. In above program a destructor can be called explicitly using following statement:

```
SAMPLE     s1;

s1.SAMPLE: :~SAMPLE();
```

7.5 Anonymous Object

We know that a constructor is called automatically whenever an object is declared. Now, if we wish to restrict this implicit call to constructor then one way is, not to declare the object. Actually, object is declared but it is hidden from the user. This type of object is known as anonymous object.

Program 43: Write a C++ program to illustrate the concept of anonymous object.

Code:

```
#include<iostream.h>

#include<conio.h>

class  SAMPLE

{

        int     n1;

        public:
            SAMPLE()
            {
                n1=5;
                cout<<"n1="<<n1<<endl;
            }
            SAMPLE(int     x)
            {
                n1=x;
                cout<<"n1="<<n1<<endl;
            }
    };
    void main()
    {
        SAMPLE();
        SAMPLE(10);
    }
```

O/P:

```
n1=5
n1=10
```

Note: Anonymous object comes into picture when constructor is called like normal function. In this case, system declares the object which is hidden from the user. Using this object the constructor is called.

7.6 main as constructor and destructor

To use main as constructor and destructor, we must define the class with class name main.

It can be understood using following example:

Example:

```
#include<iostream.h>

#include<conio.h>

class  main

{

          public:

                    main()

                    {

                              cout<<"Main as a constructor"<<endl;

                    }

                    ~main()

                    {

                              cout<<"Main as a destructor"<<endl;

                    }

};

void main()

{

          class  main  ob1;

}
```

O/P:

```
Main as a constructor

Main as a destructor
```

Note: When class is defined with class name main, It is mandatory to use class keyword while declaring the object.