CONSTRUCTORS AND DESTRUCTORS

A class encapsulates both data and functions manipulating them into a single unit. It can be further used as an abstract data type for defining a class instance known as object. As like standard data types, there must be a provision of initializing objects of a class during their definition itself. A class in C++ may contain two special member functions dealing with the internal working of a class. These functions are the constructors and the destructors. A constructor enables an object to initialize itself during creation and the destructor destroys the object when it is no longer required, by releasing all the resources allocated to it.

Definition:

A constructor is a special member function of a class whose task is to allocate the required memory as well as initialize the objects of its class. A constructor has the same name as its class. For example,

```
class sample
{
     private:
          --------------
          --------------
     public:
          sample(); //constructor
};

sample ::sample()   //note no return type required
{
    //Body of constructor if defined outside the class
}
```

A constructor has no return value specification. The constructor is such member function which is executed automatically when an object is created for that class.

When an object is created, the following process will take place.

- The object occupies space at a particular time. Instantiation of an object always involves reserving enough memory space for the data of that object.
- The instantiation does not reserve the memory for the methods. They exist only once for class, not once for every object.
- In addition to the reservation of space, the constructor may also be extended to other processes for initialization of data of that object.
- Constructor is a function (member) which is called automatically when object is created.

The C++ run time system makes sure that the constructor of a class is the first member function to be executed automatically when an object of the class is created. In other words, the constructor is executed every time an object of that class is defined. It is of course possible to define a class which has no constructor at all, in such a case, the run-time system calls a dummy constructor that performs no action when its object is created.

```
//constructor
//automatic initialization is carried out using a special
//member function.
#include<iostream.h>
#include<conio.h>



class counter
{
      private:
            int count;
      public:
            counter() {count=0;}
                  //constructor initialize value of count 0
                  //same name as class name
            void inc_count(){count++;}
            int get_count(){return count;}
};

void main()
{
      counter c1,c2;  //automatically count=0
      cout<<"\n c1="<<c1.get_count();
      cout<<"\n c2="<<c2.get_count();
      c1.inc_count();
      c2.inc_count();
      c2.inc_count();
      cout<<"\n c1="<<c1.get_count();     //c1=1
      cout<<"\n c2="<<c2.get_count();     //c2=2
      getch();
}
```

In this sample example, there is a constructor whose task is to create objects and initialize them by value count=0. The constructor counter(); is automatically invoked when object c1 and c2 of class measure are created and both objects are initialized to given value as defined by constructor.

```
//implementation of stack
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define max 20

class stack
{
      private:
            int s[max];
            int top;
      public:
            stack()   //constructor
            { top=-1;}

       void push(int x)//put number on stack
       {
```

```
                   s[++top]=x;
        }
        int pop()//take number from stack
        {

             return s[top--];
        }
};

void main()
{
      stack s;//top of stack is -1 when object s is created
      s.push(11);
      s.push(22);
      s.push(33);
      cout<<"\nNumber Popped"<<s.pop(); //33
      cout<<"\nNumber Popped"<<s.pop(); //22
      s.push(44);
      cout<<"\nNumber Popped"<<s.pop(); //44
      getch();
}
```

Types of constructor

There are three types of constructors:

1. The default constructor
2. User-defined constructor
3. Copy constructor

1. **Default constructor:** This constructor is called implicit constructor.
   - The compiler provides a (hidden) default constructor that has no arguments.
   - The default constructor takes no arguments and performs no processing other than reservation of memory.
   - This constructor is always called by compiler if no user-defined constructor is provided.
   - This constructor is automatically called while creating the object.

Example:

```
class Account
{
      private:
      ---------
      public:
      --------
};
Account acc1, acc2;    //default constructor Account()
//(hidden) is automatically called.
```

2. **User-defined constructor:**

If initialization of data of objects is required while creating an object, then the programmer has to define his own constructor for that purpose. The name for constructor should always be same as class name. The code of a user-defined constructor does not actually cause memory to be reserved for the data because it is still done by default constructor automatically. So code is concern only for initialization of data.
It is possible to have more than one constructor in a class providing that different versions are defined by different in the argument list. This gives more flexibility in the way when object of a class are instantiated. This is known as constructor overloading.

3. **Copy constructor**:

- ➢ The copy constructor creates an object as an exact copy of another object in terms of its attributes.
- ➢ The parameter of a constructor can be of any of the data types except an object of its own class as value parameter. However, a class's own object can be passed as a reference parameter. The constructor having a reference parameter is known as copy constructor.
- ➢ In copy constructor, one newly instantiated object equals another existing object of the same class.
- ➢ It uses the = (assignment) operator when creating new object. Hence newly created object inherit the attributes of existing object.

```
class Test
{
     private:
          ---------
          ---------
     public:
          Test(Test &t);
               //copy constructor, reference object as parameter
          --------
 };
Test T1; //object T1 created, default constructor
Test T2=T1;  //copy constructor
//Alternative
 Test T2(T1);
```

Calling the constructor:

Default

- Whenever an object is instantiated, a default constructor is called automatically to reserve the memory for that object.
- A default constructor is called if no user-defined constructor is provided in class definition and it allocates memory for objects.
- It is not possible to pass any parameters to the default constructor and to make it perform or call any other process.
  Account acc1, acc2; //default constructor is called.

User-defined

- The main advantage of user-defined constructor is to initialize the object while it is created.
- A user-defined constructor have following special characteristics
  - ❖ It takes same name as that of class.
  - ❖ Should be declared in the public section of class definition.
  - ❖ They are invoked automatically when the object are created.
  - ❖ May take argument but do not have any return type.

Copy constructor:

- Called using assignment operator or object as arguments automatically.

Parameterized Constructor:

Constructors can be invoked with arguments, just as in the case of functions. The argument list can be specified within braces similar to the argument-list in the function. So a user-defined constructor having arguments is called parameterized constructor. While creating object, we should pass argument for the parameterized constructor. Since C++ allows function overloading, a constructor with arguments can co-exist with another constructor without argument.

```
class Test
{
        private:
                ---------
                ---------
        public:
                Test(int data1)//constructor with parameter
                {
                        --------------
                }
                --------
 };
Test t1(4); //4 is passed as parameter
Test t2=5;  //5 is passed as parameter
```

Example:

```
//example of parameterized constructor
#include<iostream.h>
#include<conio.h>
class BankAccount
{
        private:
                        int accno;
                        float bal;
         public:
                BankAccount(int , float );//constructor
                void display()
                {
                        cout<"Account Info";
```

```
                  cout<<"\n Account No:"<<accno;
                  cout<<"\n Balance:"<<bal;
            }

};

//definition of parameterized constructor
BankAccount :: BankAccount(int ac, float b)
      {     accno = ac;  bal = b;  }



main()
{
      BankAccount acc1(1024, 5000.0);    //constructor called implicity
      acc1.display();
      BankAccount acc2 = BankAccount(2024, 5000.0); //called explicity
      acc2.display();
      getch();
}
```

Constructor overloading:

A class can have multiple constructors. If more than one constructors are used in a class, it is known as constructor overloading. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

```
//constructor overloading
#include <iostream.h>
#include <conio.h>

class Account
{
       private:
            int accno;   float balance;
      public:
            Account()   //constructor1
            { accno=1024,balance=5000.55;}


            Account(int acc)   //constructor2 with one argument
            {
                  accno=acc;
                  balance=0.0;
            }

            Account(int acc, float bal)//constructor3,with two arguments
            {
                  accno=acc; balance=bal;
            }
            void display()
            {
                  cout<<"Account no.="<<accno<<endl;
                  cout<<"Balance="<<balance<<endl;
            }
 };  //end of class definition

 void main()
 {
      Account acc1;  //constructor1
      Account acc2(100);  //constructor2
      Account acc3(200, 8000.50);  //constructor3
      cout<<endl<<"Account information"<<endl;
      acc1.display();
      acc2.display();
      acc3.display();
      cout<<"Account information"<<endl;
      acc1.display();
      acc2.display();
      acc3.display();
   getch();
 } //end of main()
```

Another Example

```
//User-defined copy constructor
#include<iostream.h>
#include<conio.h>
class Code
{
       private:
            int id;
      public:
            Code(){}
            Code(int a) {id=a;}
            Code(Code &x) //reference object argument
            { id = x.id;  }

            int display()
            {     return id; }
}; //end of class definition

void main()
{
  Code p1(55);
  Code p2(p1);  //copy constructor
  Code p4=p2;
  Code p3=p1;
  cout<<"\n Code p1="<<p1.display();
  cout<<"\n Code p2="<<p2.display();
  cout<<"\n Code p3="<<p3.display();
  cout<<"\n Code p4="<<p4.display();
  getch();
}
// All prints 55
```

Destructors

When an object is created the constructed is called automatically. Like wise a class can have another special member function called the destructor. When an object is no longer needed it can be destroyed. Destructor is invoked when an object is destroyed. This function complements the operation performed by any of the constructors. When a program is interrupted using an exit( ) call, the destructor are called for all objects which exist at that time. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~).

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 to clean up storage that is no longer accessible. It may be a good practice to declare destructors in a program since it releases memory space for future use.

```
class Test
{
     private:
          ---------
          ---------
     public:
          Test()//constructor
          { }
          ~Test()//destructor
          {}
 };
```

Example:

```
//example of destructor
#include<iostream.h>
#include<conio.h>
class Test
{
      public:
      Test()//constructor
      {
      cout<<"\nControl is in constructor";
      }
      ~Test()//destructor
      {
      cout<<"\n Control is in destructor";
      }
};

void main()
{
      Test t; //constructor is called
      cout<<"Function main() terminating...";
      //object t goes out of scope, destructor is called
}
```