

CONSTRUCTORS AND DESTRUCTORS

Constructors and destructors are special member functions which performs the task of initialisation and clean up respectively for the safety purposes.

First let's talk about constructors-

What is constructor?

It is a special member function of every class (whether you define it or not) which is invoked (called) automatically by C++ compiler.

Why it is called by C++ compiler?

It is invoked because it is mainly used to initialise the data members of the class. In other words, constructor is used to initialise the object of the class.

Example,

```

1  #include <iostream>
2  using namespace std;
3  class student
4  {
5      string name;
6      int roll_no;
7      float marks;
8      public:
9          void show()
10         {
11             cout<<"Name          : "<<name<<endl;
12             cout<<"Roll number : "<<roll_no<<endl;
13             cout<<"Marks       : "<<marks<<endl;
14         }
15     }student1;
16
17     int main()
18     {
19         student1.show();
20         return 0;
21     }

```

Default constructor
will be called

F:\e drive\Dev-Cpp\2019-2020 C++\constructor1.exe

```

Name          :
Roll number :0
Marks       :0
-----
Process exited after 0.2053 seconds with return value 0
Press any key to continue . . .

```

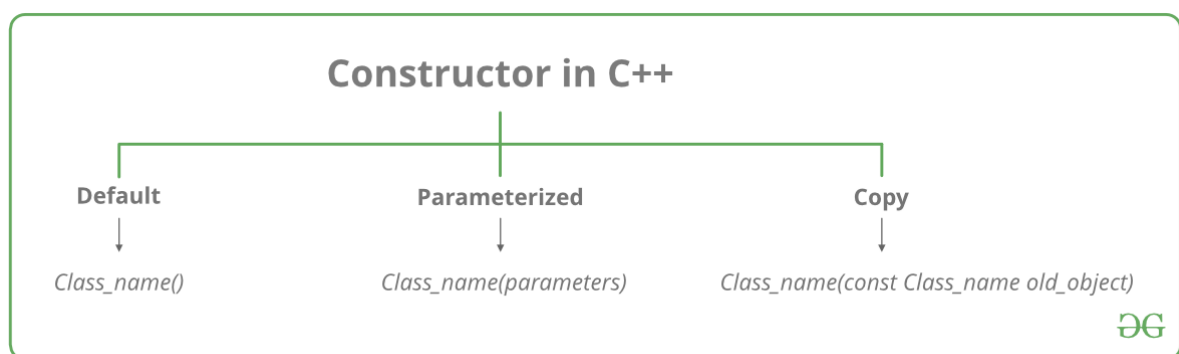
In the above program, the data members of class has been initialised by the default constructor.

How constructor functions are different from normal functions of class?

1. The purpose of constructor is to create the instance of a class whereas the purpose of normal method is used to express the behaviour of an object.
2. Constructors do not have any return type whereas normal methods of a class do have a return type i.e. void or non-void type.
3. Constructors have same name as class name whereas normal methods don't have same name as class name.
4. Constructors are called implicitly by C++ compiler and can be called explicitly as well whereas normal methods are always called explicitly using already created object and dot operator.

Types of constructors

1. Default constructor
2. Parameterised constructor
3. Copy constructor



Default constructor

It is the constructor which do not accepts any arguments/parameters and is called automatically by the compiler by default if it is not added in the program by the programmer. Hence, the name is default constructor.

Syntax:

```
/*.....class with constructor.....*/
class class_name
{
    .....
public:
    class_name(); //constructor declared or constructor prototype
    .....
```

```
};
class_name :: class_name() //constructor defined
{
    //constructor function body
}
```

A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values.

Example 1:

```
class ex
{
    int x,y;
    public:
    ex(int x=0, int y=0) //default constructor with default values
    {}
};
```

Example 2:

```
1 class test
2 {
3     int m,n;
4     public:
5         test()
6         {
7             m=0;
8             n=0;
9         }
10 }
11 int main()
12 {
13     test t1; //calls default constructor
14     return 0;
15 }
```

If no user-defined constructor exists for a class A and one is needed, the compiler implicitly declares a default parameterless constructor A::A(). This constructor is an inline public member of its class. The compiler will implicitly define A::A() when the compiler uses this constructor to create an object of type A. The constructor will have no constructor initializer and a null body.

No default constructor is created for a class that has any constant or reference type members.

```
#include <iostream>
using namespace std;
class student
```

```

{
    string name;
    int roll_no;
    float marks;
    public:
        student() //default construtor
        {
            cout<<"Object created\n";
            name  = " ";
            roll_no= 0;
            marks = 0;
        }
        void show()
        {
            cout<<"Name      :"<<name<<endl;
            cout<<"Roll number :"<<roll_no<<endl;
            cout<<"Marks      :"<<marks<<endl;
        }
        ~student() //destructor
        {
            cout<<"Object destroyed..\n";
        }
}student1; //constructor will be called automatically

int main()
{
    student1.show();
    return 0;
}

```

```

Object created
Name      :
Roll number :0
Marks      :0
Object destroyed..

-----
Process exited after 0.1596 seconds with return value 0
Press any key to continue . . .

```

```

#include <iostream>
using namespace std;
class DemoDC {
    private:
    int num1, num2 ;
    public:
    DemoDC() {
        num1 = 10; //default constructor
        num2 = 20;
    }
    void display() {
        cout<<"num1 = "<< num1 <<endl;
        cout<<"num2 = "<< num2 <<endl;
    }
};
int main() {
    DemoDC obj;
    obj.display();
    return 0;
}

```

Output

num1 = 10

num2 = 20

Parameterised constructor

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Example:

```

- student(string n,int r,float m) //parameterised constructor
{
    cout<<"Object created\n";
    name=n;
    roll_no= r;
    marks= m;
}

```

Sample Program:

```


#include <iostream>
using namespace std;
class student
{
    string name;
    int roll_no;
    float marks;
public:
    student() //constructor
    {
        cout<<"Object created\n";
        name = " ";
        roll_no= 0;
        marks = 0;
    }
    student(string n,int r,float m) //parameterised constructor
    {
        cout<<"Object created\n";
        name=n;
        roll_no= r;
        marks= m;
    }
    void show()
    {
        cout<<"Name      :"<<name<<endl;
        cout<<"Roll number :"<<roll_no<<endl;
        cout<<"Marks      :"<<marks<<endl;
    }

    ~student()
    {
        cout<<"Object destroyed..\n";
    }
};

int main()
{
    student student1; //call default constructor
    student1.show();
    student student2("Rohit",12,86.3); //call parameterised constructor
    student2.show();
    return 0;
}

```

}

 F:\e drive\Dev-Cpp\2019-2020 C++\constructor4.exe

```

Object created
Name      :
Roll number :0
Marks     :0
Object created
Name      :Rohit
Roll number :12
Marks     :86.3
Object destroyed..
Object destroyed..

-----
Process exited after 0.06391 seconds with return value 0
Press any key to continue . . .

```

Program to find number of objects created

```

#include <iostream>
using namespace std;
class COUNT
{
    static int x;
    public:
        COUNT()
        {
            ++x;
        }
        void show()
        {
            cout<<"Number of objects created ="<<x;
        }
};
int COUNT::x;
int main()
{
    int n;
    cout<<"Enter number of objects-\n";
    cin>>n;
    COUNT obj[n];
    obj[0].show();
    return 0;
}

```

F:\e drive\Dev-Cpp\2019-2020 C++\numberofobjects.exe

```
Enter number of objects-
10
Number of objects created =10
-----
Process exited after 2.99 seconds with return value 0
Press any key to continue . . .
```

COPY CONSTRUCTOR

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.

Program with all types of constructor

```
#include<iostream>
using namespace std;
class example
{
    int x;
    int y;
    public:
        example()
        {
            x=10;
            y=20;
        }
}
```



```

        example(int a,int b)
        {
            x=a;
            y=b;
        }
        example(const example &obj1)
        {
            x= obj1.x;
            y= obj1.y;
        }
        void show()
        {
            cout<<endl;
            cout<<"x= "<<x<<" ";
            cout<<"y= "<<y;
        }
    };
    int main()
    {

        example e1; //call to default constructor
        e1.show();
        example e2(20,30); //call to parameterised constructor
        e2.show();
        example e3 =e2; //call to copy constructor
        e3.show();
        return 0;
    }

```

```

x= 10 y= 20
x= 20 y= 30
x= 20 y= 30
-----
Process exited after 0.04734 seconds with return value 0
Press any key to continue . . .

```

NEED TO DEFINE USER-DEFINED COPY CONSTRUCTOR

C++ compiler provide default copy constructor (and assignment operator) with class. When we don't provide implementation of copy constructor (and assignment operator) and tries to initialize object with already initialized object of same class then copy constructor gets called and copies members of class one by one in target object.

The problem with default copy constructor (and assignment operator) is – When we have members which dynamically gets initialized at run time, default copy constructor copies this members with address of dynamically allocated memory and not real copy of this memory. Now both the objects points to the same memory and changes in one reflects in another object, Further the main disastrous effect is, when we delete one of this object other object still points to same memory, which will be dangling pointer, and memory leak is also possible problem with this approach.

Hence, in such cases, we should always write our own copy constructor

Program with default copy constructor

```
#include <iostream>
#include<cstring>
using namespace std;
class student{

    char *name;
    int len;
    int roll_no;
    float marks;
public:
    student(const char* s,int r,float m) //constructor
    {
        len= strlen(s);
        name= new char[len+1];
        strcpy(name,s);
        roll_no=r;
        marks=m;
    }
    void show()
    {
        cout<<"Name    :"<<name<<endl;
        cout<<"Roll number :"<<roll_no<<endl;
        cout<<"Marks    :"<<marks<<endl<<endl;
    }
    void change(const char* str)
    {
        strcpy(name,str);
    }

};

int main()
```

```

{
    student student1("Harman",11,89);
    cout<<"Values in object 1\n";
    student1.show();
    student student2= student1;
    cout<<"\nAfter Copying..\nValues in object 2\n";
    student2.show();
    student2.change("Manav");

    cout<<"\nAfter calling change function for object 2\n";
    cout<<"Values in object 2\n";
    student2.show();
    cout<<"Values in object 1\n";
    student1.show();
    return 0;
}

```

```

Values in object 1
Name           :Harman
Roll number    :11
Marks          :89

After Copying..
Values in object 2
Name           :Harman
Roll number    :11
Marks          :89

After calling change function for object 2
Values in object 2
Name           :Manav
Roll number    :11
Marks          :89

Values in object 1
Name           :Manav
Roll number    :11
Marks          :89

```

Program with user-defined copy constructor

```
#include <iostream>
```

```

#include<cstring>
using namespace std;
class student{

    char *name;
    int len;
    int roll_no;
    float marks;
public:
    student(const char* s,int r,float m) //construtor
    {
        len= strlen(s);
        name= new char[len+1];
        strcpy(name,s);
        roll_no=r;
        marks=m;
    }
    void show()
    {
        cout<<"Name    :"<<name<<endl;
        cout<<"Roll number :"<<roll_no<<endl;
        cout<<"Marks    :"<<marks<<endl<<endl;
    }
    void change(const char* str)
    {
        strcpy(name,str);
    }
    student(const student &old_obj)
    {
        len=old_obj.len;
        name= new char[len+1];
        strcpy(name,old_obj.name);
        roll_no= old_obj.roll_no;
        marks = old_obj.marks;
    }

};

int main()
{
    student student1("Harman",11,89);
    cout<<"Values in object 1\n";
    student1.show();
    student student2= student1;
}

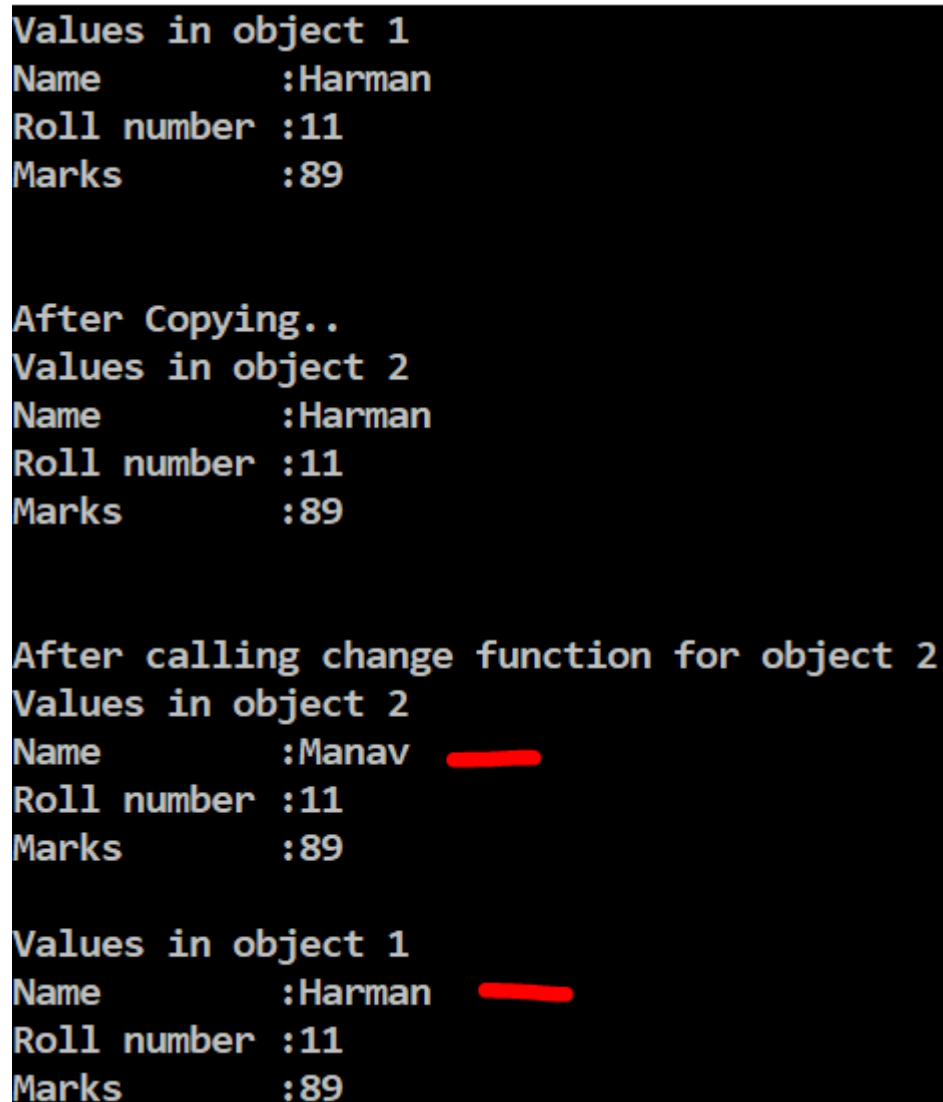
```

```

        cout<<"\nAfter Copying..\nValues in object 2\n";
        student2.show();
        student2.change("Manav");

        cout<<"\nAfter calling change function for object 2\n";
        cout<<"Values in object 2\n";
        student2.show();
        cout<<"Values in object 1\n";
        student1.show();
        return 0;
}

```



The screenshot shows the output of a C++ program. It displays the state of two objects, object 1 and object 2, at different stages of execution. Object 1's data (Name: Harman, Roll number: 11, Marks: 89) remains constant. Object 2's data is shown before and after a change function is called, demonstrating that the change only affects object 2 and not object 1.

```

Values in object 1
Name           :Harman
Roll number    :11
Marks          :89

After Copying..
Values in object 2
Name           :Harman
Roll number    :11
Marks          :89

After calling change function for object 2
Values in object 2
Name           :Manav
Roll number    :11
Marks          :89

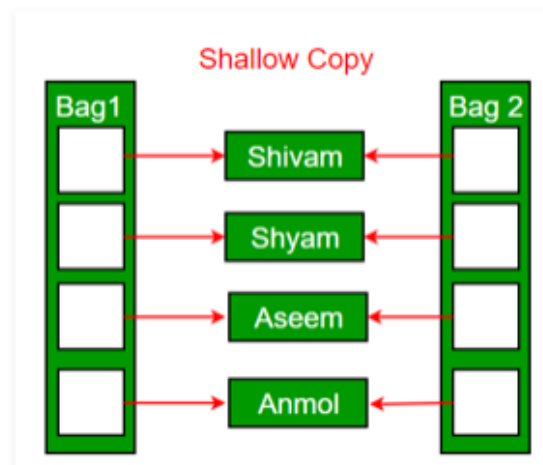
Values in object 1
Name           :Harman
Roll number    :11
Marks          :89

```

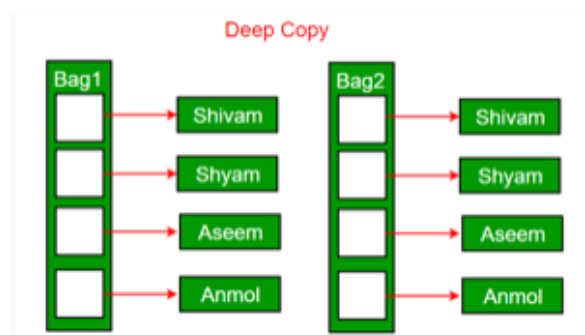
What would be the problem if we remove copy constructor from above code?

If we remove copy constructor from the above program, we don't get the expected output. The changes made to name for second object would reflect in first object as well which is never expected.

Default constructor does only shallow copy.



Deep copy is possible only with user defined copy constructor. In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



What is destructor?

Destructor is basically opposite of constructor. It is used to deallocate memory of an object which was allocated by the constructor function. That's why, the syntax is:

~ constructor

i.e. ~ classname() where ~ tilde symbol signifies the complement or negation.

Moreover, destructors cannot be overloaded as they don't accept any parameters.