C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1;  //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
class Student
 {
     public:
     int id;  //field or data member
```

```
      float salary; //field or data member
      String name;//field or data member
  }
```

C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
#include <iostream>
using namespace std;
class Student {
   public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
};
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

Output:

```
201
Sonoo Jaiswal
```

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
#include <iostream>
using namespace std;
class Student {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     void insert(int i, string n)
      {
         id = i;
         name = n;
      }
     void display()
      {
         cout<<id<<"  "<<name<<endl;
      }
};
int main(void) {
   Student s1; //creating an object of Student
   Student s2; //creating an object of Student
   s1.insert(201, "Sonoo");
   s2.insert(202, "Nakul");
   s1.display();
   s2.display();
   return 0;
}
```

Output:

```
201  Sonoo
202  Nakul
```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
#include <iostream>
using namespace std;
class Employee {
  public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
      float salary;
      void insert(int i, string n, float s)
       {
          id = i;
          name = n;
          salary = s;
       }
      void display()
       {
          cout<<id<<"  "<<name<<"  "<<salary<<endl;
       }
};
int main(void) {
```

```
Employee e1; //creating an object of Employee
Employee e2; //creating an object of Employee
e1.insert(201, "Sonoo",990000);
e2.insert(202, "Nakul", 29000);
e1.display();
e2.display();
return 0;
}
```

Output:

```
201  Sonoo  990000
202  Nakul  29000
```

In C++, there are three access specifiers:

- `public` - members are accessible from outside the class
- `private` - members cannot be accessed (or viewed) from outside the class
- `protected` - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

Example

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};
```

```
int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

Output

```
prog.cpp: In function ‘int main()’:
prog.cpp:14:9: error: ‘int MyClass::y’ is private within this context
   14 |   myObj.y = 50;  // Not allowed (y is private)
      |         ^
prog.cpp:8:9: note: declared private here
    8 |     int y;   // Private attribute
      |         ^

prog.cpp: In function ‘int main()’:
prog.cpp:14:9: error: ‘int MyClass::y’ is private within this context
   14 |   myObj.y = 50;  // Not allowed (y is private)
      |         ^
prog.cpp:8:9: note: declared private here
    8 |     int y;   // Private attribute
      |
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

In brief, A particular procedure called a constructor is called automatically when an object is created in C++. In general, it is employed to create the data members of new things. In C++, the class or structure name also serves as the constructor name. When an object is completed, the constructor is called. Because it creates the values or gives data for the thing, it is known as a constructor.

The Constructors prototype looks like this:

```
<class-name> (list-of-parameters);
```

The following syntax is used to define the class's constructor:

```
<class-name> (list-of-parameters) { // constructor definition }
```

The following syntax is used to define a constructor outside of a class:

```
<class-name>: :<class-name> (list-of-parameters){ // constructor definition}
```

Constructors lack a return type since they don't have a return value.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
```

```
        cout<<"Default Constructor Invoked"<<endl;
    }
};
int main(void)
{
  Employee e1; //creating an object of Employee
  Employee e2;
  return 0;
}
```

Output:

```
Default Constructor Invoked

Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
class Car {        // The class
 public:          // Access specifier
   string brand;  // Attribute
string model;  // Attribute
   int year;      // Attribute
Car(string x, string y, int z) { // Constructor with parameters
        brand = x;
        model = y;
```

```
  year = z;

         }

};

int main() {

  // Create Car objects and call the constructor with different values

  Car carObj1("BMW", "X5", 1999);

  Car carObj2("Ford", "Mustang", 1969);

  // Print values

  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";

  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";

  return 0;

}
```

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution `::` operator, followed by the name of the constructor (which is the same as the class):

```
class Car {        // The class
  public:          // Access specifier
string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
          Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}
```

```
int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

What distinguishes constructors from a typical member function?

1. Constructor's name is the same as the class's
2. Default There isn't an input argument for constructors. However, input arguments are available for copy and parameterized constructors.
3. There is no return type for constructors.
4. An object's constructor is invoked automatically upon creation.
5. It must be shown in the classroom's open area.
6. The C++ compiler creates a default constructor for the object if a constructor is not specified (expects any parameters and has an empty body).

By using a practical example, let's learn about the various constructor types in C++. Imagine you visited a store to purchase a marker. What are your alternatives if you want to buy a marker? For the first one, you ask a store to give you a marker, given that you didn't specify the brand name or colour of the marker you wanted, simply asking for one amount to a request. So, when we just said, "I just need a marker," he would hand us whatever the most popular marker was in the market or his store. The default constructor is exactly what it sounds like! The second approach is to go into a store and specify that you want a red marker of the XYZ brand. He will give you that marker since you have brought up the subject. The parameters have been set in this instance thus. And a parameterized constructor is exactly what it sounds like! The third one requires you to visit a store and declare that you want a marker that looks like this (a physical marker on your hand). The shopkeeper will thus notice that marker. He will provide you with a new marker when you say all right. Therefore, make a copy of that marker. And that is what a copy constructor does!

What are the characteristics of a constructor?

1. The constructor has the same name as the class it belongs to.
2. Although it is possible, constructors are typically declared in the class's public section. However, this is not a must.
3. Because constructors don't return values, they lack a return type.
4. When we create a class object, the constructor is immediately invoked.
5. Overloaded constructors are possible.
6. Declaring a constructor virtual is not permitted.
7. One cannot inherit a constructor.
8. Constructor addresses cannot be referenced to.
9. When allocating memory, the constructor makes implicit calls to the new and delete operators.

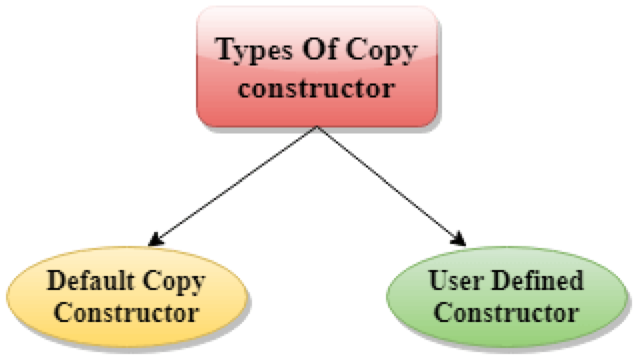
C++ Copy Constructor

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.

- **User Defined constructor:** The programmer defines the user-defined constructor.



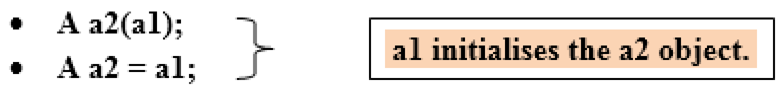
Syntax Of User-defined Copy Constructor:

```
Class_name(const class_name &old_object);
```

Consider the following situation:

```
class A
{
    A(A &x) //  copy constructor.
   {
       // copyconstructor.
   }
}
```

In the above case, **copy constructor can be called in the following ways:**



- A a2(a1);
- A a2 = a1;

a1 initialises the a2 object.

Let's see a simple example of the copy constructor.

// program of the copy constructor.

```
#include <iostream>
using namespace std;
class A
{
  public:
   int x;
   A(int a)                // parameterized constructor.
   {
     x=a;
   }
   A(A &i)               // copy constructor
   {
       x = i.x;
   }
};
int main()
{
 A a1(20);               // Calling the parameterized constructor.
A a2(a1);                //  Calling the copy constructor.
cout<<a2.x;
 return 0;
}
```

Output:

```
20
```

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

- When we initialize the object with another existing object of the same class type. For example, Student s1 = s2, where Student is the class.
- When the object of the same class type is passed by value as an argument.
- When the function returns the object of the same class type by value.

Two types of copies are produced by the constructor:

- Shallow copy
- Deep copy

Shallow Copy

- The default copy constructor can only produce the shallow copy.
- A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.

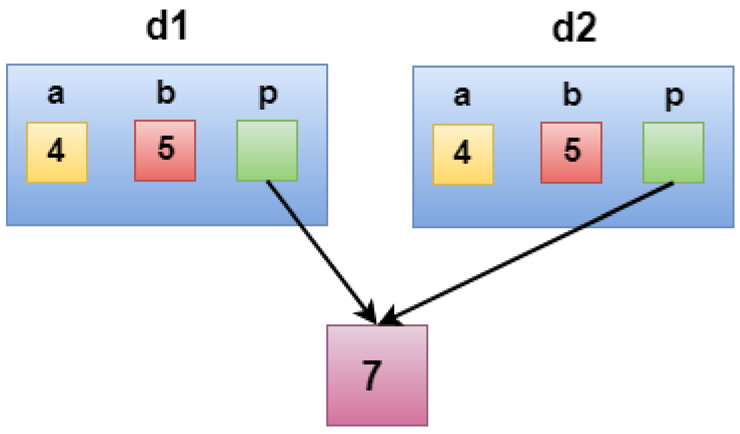
Let's understand this through a simple example:

```
#include <iostream>

using namespace std;

class Demo
{
    int a;
    int b;
    int *p;
    public:
    Demo()
    {
        p=new int;
    }
    void setdata(int x,int y,int z)
    {
        a=x;
        b=y;
        *p=z;
    }
    void showdata()
    {
        std::cout << "value of a is : " <<a<< std::endl;
        std::cout << "value of b is : " <<b<< std::endl;
        std::cout << "value of *p is : " <<*p<< std::endl;
    }
};
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
  Demo d2 = d1;
```

```
d2.showdata();
  return 0;
}
```

Output:

```
value of a is : 4

value of b is : 5

value of *p is : 7
```



In the above case, a programmer has not defined any constructor, therefore, the statement **Demo d2 = d1;** calls the default constructor defined by the compiler. The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer p of both the objects point to the same memory location. Therefore, when the memory of a field is freed, the memory of another field is also automatically freed as both the fields point to the same memory location. This problem is solved by the **user-defined constructor** that creates the **Deep copy**.

Deep copy

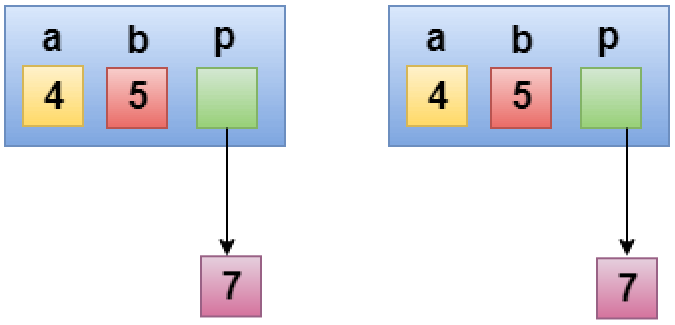
Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations. In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

Let's understand this through a simple example.

```
#include <iostream>
using namespace std;
class Demo
{
    public:
    int a;
    int b;
    int *p;

    Demo()
    {
        p=new int;
    }
    Demo(Demo &d)
    {
        a = d.a;
        b = d.b;
        p = new int;
        *p = *(d.p);
    }
    void setdata(int x,int y,int z)
    {
        a=x;
        b=y;
        *p=z;
    }
    void showdata()
    {
        std::cout << "value of a is : " <<a<< std::endl;
        std::cout << "value of b is : " <<b<< std::endl;
        std::cout << "value of *p is : " <<*p<< std::endl;
    }
};
int main()
{
  Demo d1;
  d1.setdata(4,5,7);
  Demo d2 = d1;
  d2.showdata();
  return 0;
}
```

Output:

```
value of a is : 4

value of b is : 5

value of *p is : 7
```



In the above case, a programmer has defined its own constructor, therefore the statement **Demo d2 = d1;** calls the copy constructor defined by the user. It creates the exact copy of the value types data and the object pointed by the pointer p. Deep copy does not create the copy of a reference type variable.

What is a destructor in C++?

An equivalent special member function to a constructor is a destructor. The constructor creates class objects, which are destroyed by the destructor. The word "destructor," followed by the tilde () symbol, is the same as the class name. You can only define one destructor at a time. One method of destroying an object made by a constructor is to use a destructor. Destructors cannot be overloaded as a result. Destructors don't take any arguments and don't give anything back. As soon as the item leaves the scope, it is immediately called. Destructors free up the memory used by the objects the constructor generated. Destructor reverses the process of creating things by destroying them.

The language used to define the class's destructor

```
1. ~ <class-name>()
2.         {
3.          }
```

The language used to define the class's destructor outside of it

```
1. <class-name>: : ~ <class-name>(){}
```

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Constructor Invoked"<<endl;
        }
        ~Employee()
        {
            cout<<"Destructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

Output:

```
Constructor Invoked

Constructor Invoked

Destructor Invoked

Destructor Invoked
```