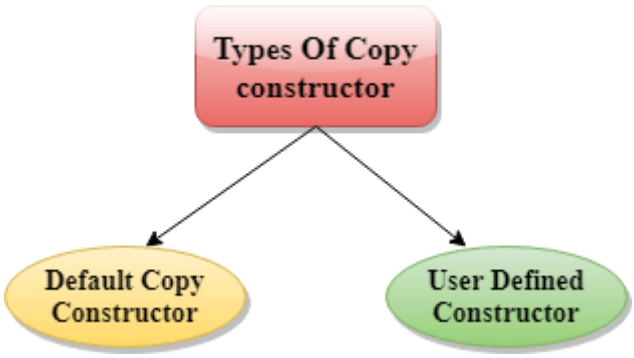
Lecture 5: C++ Copy Constructor

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:

1. Class_name(const class_name &old_object);

Consider the following situation:

```
1. class A
2. {
3.    A(A &x) //  copy constructor.
4.   {
5.       // copyconstructor.
6.   }
7. }
```

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.

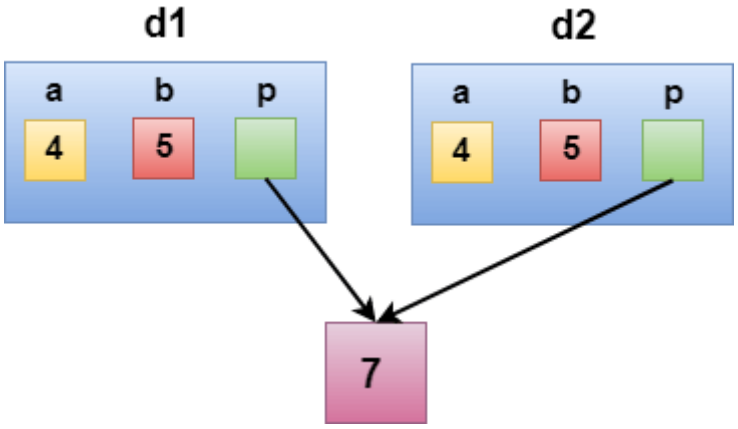
Let's understand this through a simple example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. class Demo
6. {
7.    int a;
8.    int b;
9.    int *p;
10.         public:
11.         Demo()
12.         {
13.            p=new int;
14.         }
15.         void setdata(int x,int y,int z)
16.         {
```

```
17.             a=x;
18.             b=y;
19.             *p=z;
20.         }
21.         void showdata()
22.         {
23.             std::cout << "value of a is : " <<a<< std::endl;
24.             std::cout << "value of b is : " <<b<< std::endl;
25.             std::cout << "value of *p is : " <<*p<< std::endl;
26.         }
27.     };
28.     int main()
29.     {
30.       Demo d1;
31.       d1.setdata(4,5,7);
32.       Demo d2 = d1;
33.       d2.showdata();
34.        return 0;
35.     }
```

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.

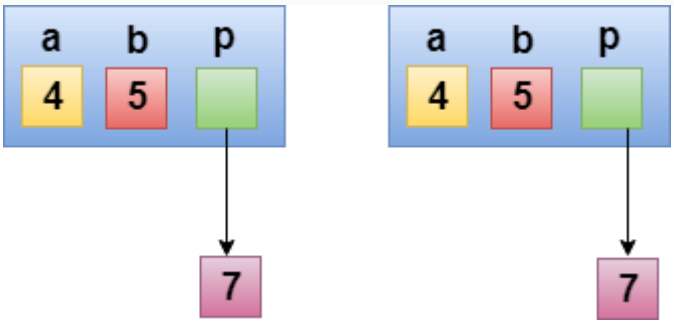
Let's understand this through a simple example.

```
1. #include <iostream>
2. using namespace std;
3. class Demo
4. {
5.    public:
6.    int a;
7.    int b;
8.    int *p;
9.
10.          Demo()
11.          {
12.             p=new int;
13.          }
14.          Demo(Demo &d)
15.          {
16.             a = d.a;
17.             b = d.b;
18.             p = new int;
19.             *p = *(d.p);
20.          }
21.          void setdata(int x,int y,int z)
22.          {
23.             a=x;
24.             b=y;
```

```
25.             *p=z;
26.         }
27.         void showdata()
28.         {
29.             std::cout << "value of a is : " <<a<< std::endl;
30.             std::cout << "value of b is : " <<b<< std::endl;
31.             std::cout << "value of *p is : " <<*p<< std::endl;
32.         }
33.     };
34.     int main()
35.     {
36.       Demo d1;
37.       d1.setdata(4,5,7);
38.       Demo d2 = d1;
39.       d2.showdata();
40.       return 0;
41.     }
```

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.