FAST NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, PESHAWAR

DEPARTMENT OF COMPUTER SCIENCE

CL217 – OBJECT ORIENTED PROGRAMMING LAB

Dynamic Memory Allocation and Copy Constructor

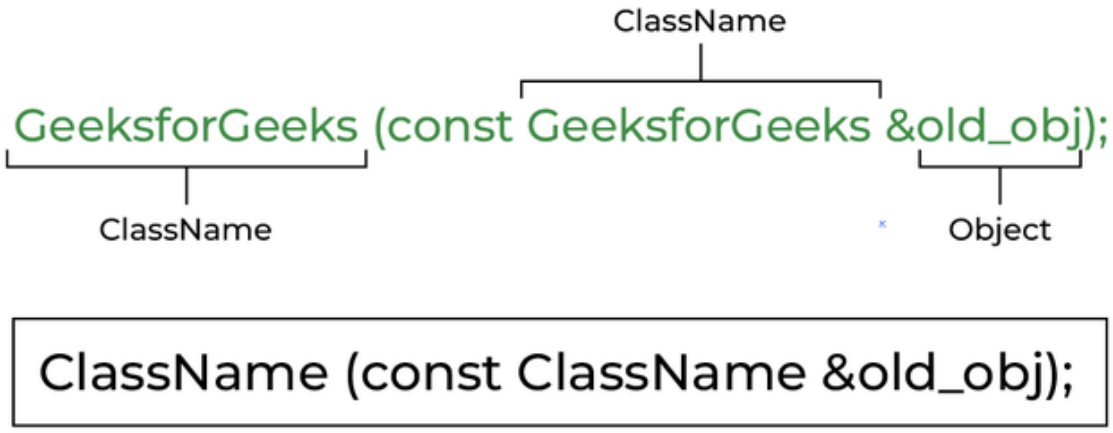
LAB MANUAL # 08

Instructor: Mazhar Iqbal

SEMESTER SPRING 2023

Copy Constructor

A **copy constructor** is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.
Example:



Syntax of Copy Constructor

Characteristics of Copy Constructor

1. The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
2. Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)

{

       id=t.id;

}
```

3. The process of initializing members of an object through a copy constructor is known as ***copy initialization.***
4. It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.
5. The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Types of Copy Constructors

1. Default Copy Constructor

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

- C++

```cpp
// Implicit copy constructor Calling
#include <iostream>
using namespace std;

class Sample {
    int id;

public:
    void init(int x) { id = x; }
    void display() { cout << endl << "ID=" << id; }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    // Implicit Copy Constructor Calling
    Sample obj2(obj1); // or obj2=obj1;
    obj2.display();
    return 0;
}
```

Output

```
ID=10

ID=10
```

2. User Defined Copy Constructor

A user-defined copy constructor is generally needed when an object owns pointers or non-shareable references, such as to a file, in which case a destructor and an assignment operator should also be written

- C++

```
// C++ program to demonstrate the working
// of a COPY CONSTRUCTOR
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    // Copy constructor
    Point(const Point& p1)
    {
        x = p1.x;
        y = p1.y;
    }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX()
```

```
         << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX()
         << ", p2.y = " << p2.getY();
    return 0;
}
```

Output

```
p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15
```

When is the copy constructor called?

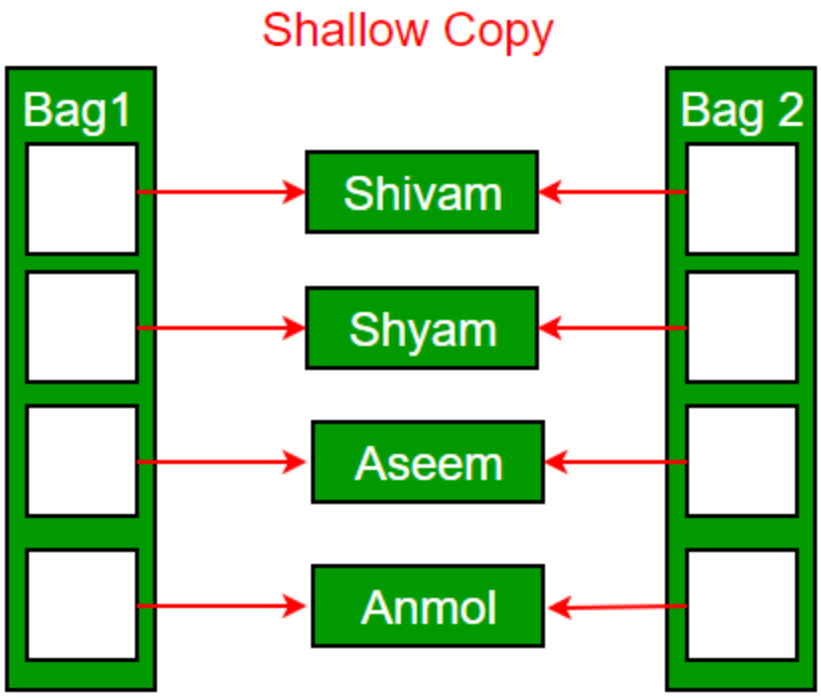
In C++, a Copy Constructor may be called in the following cases:

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

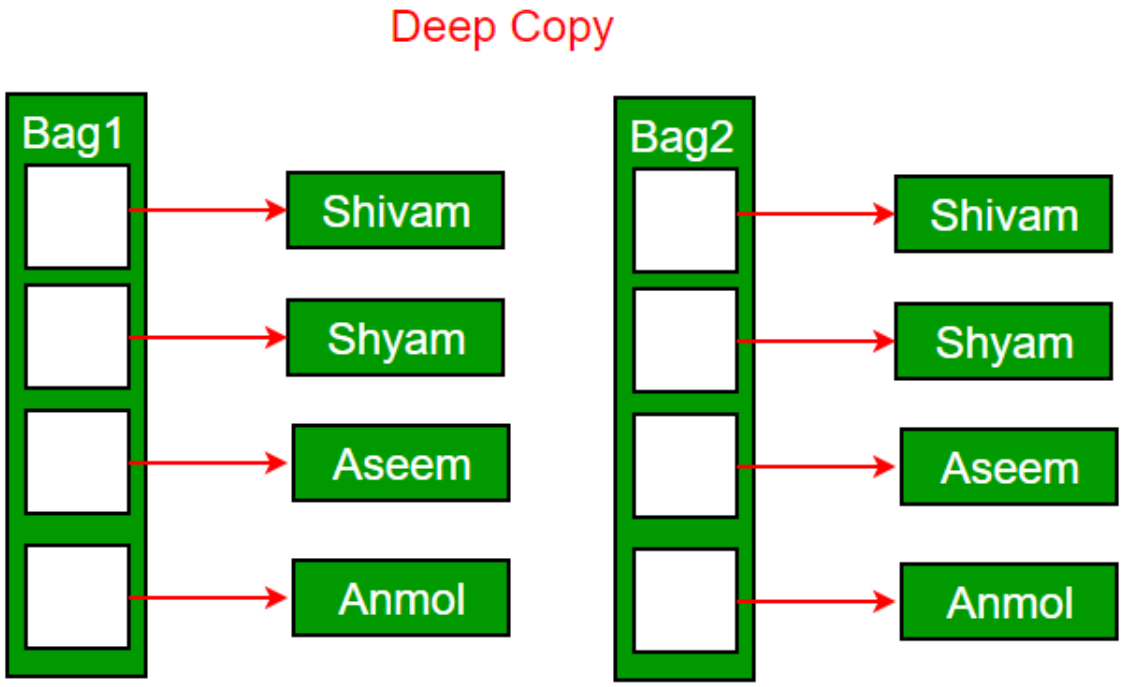
When is a user-defined copy constructor needed?

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler-created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like *a file handle*, a network connection, etc.

The default ***constructor does only shallow copy.***



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



Copy constructor vs Assignment Operator

The main difference between Copy Constructor and Assignment Operator is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

Which of the following two statements calls the copy constructor and which one calls the assignment operator?

```
MyClass t1, t2;

MyClass t3 = t1;

t2 = t1;
```

A copy constructor is called when a new object is created from an existing object, as a copy of the existing object. The assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls the copy constructor and (2) calls the assignment operator. See this for more details.

Example Code:

In the following code, there is a pointer as a data member. When the default copy constructor is called, basically the address of the data member radius is assigned to the data member of the other object. So, ultimately, the radius of both objects will point to the same memory. It is known as a shallow copy. Any changes made to the radius from object 1 (c), will reflect in c2.

```
#include<iostream>
using namespace std;
class Circle
{
	int *radius;
	public:

		void setRadius(int r)
		{
			*radius=r;
		}
		void display()
		{
			cout<<*radius;
		}
		Circle()
		{
			radius=new int;
		}
};
int main()
{
	Circle c;
	c.setRadius(5);
	cout<<"\nC1: Radius:";
	c.display();
	Circle c2(c);
	cout<<"\nC2: Radius:";
	c2.display();
	c.setRadius(6);
	cout<<"\nC1: Radius:";
	c.display();
	cout<<"\nC2: Radius:";
	c2.display();
}
```

Output:

C1: Radius:5

C2: Radius:5

C1: Radius:6

C2: Radius:6

To solve this issue, we have to change the behavior of the default copy constructor by defining our own copy constructor.

Here is the correct way of doing so.

```
#include<iostream>
using namespace std;
class Circle
{
   int *radius;
public:
   Circle()
   {
      radius = new int(0);
   }

   Circle(const Circle &c)
   {
      cout << "\nCopy constructor called\n";
      radius = new int(*(c.radius));
   }

   ~Circle()
   {
      cout << "\nDestructor called" << endl;
      delete radius;
   }

   void setRadius(int r)
   {
      *radius = r;
   }

   void display()
   {
      cout << *radius;
   }
};

int main()
{
            Circle c;
            c.setRadius(5);
            cout<<"\nC1: Radius:";
            c.display();
            Circle c2(c);
            cout<<"\nC2: Radius:";
            c2.display();
            c.setRadius(6);
            cout<<"\nC1: Radius:";
            c.display();
            cout<<"\nC2: Radius:";
            c2.display();
}
```

Output:

C1: Radius:5

Copy constructor called

C2: Radius:5

C1: Radius:6

C2: Radius:5

Destructor called

Destructor called

Creating a Dynamic array in a Class with a Copy Constructor

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

private:
   int* arr;
   int arrSize;
public:
   DynamicArray(int size) {
      arrSize = size;
      arr = new int[arrSize];
   }
   DynamicArray(DynamicArray &d) {
         cout<<"\nCopy constructor called"<<endl;
      arr = new int[d.getSize()];
      arrSize=d.getSize();
      for(int i=0; i<d.getSize(); i++)
      {
         arr[i]=d.arr[i];
       }
   }

   ~DynamicArray() {
         cout<<"\nDestructor called"<<endl;
      delete[] arr;
   }

         void printArray()
         {
                  for(int i=0; i<arrSize; i++)
                  {
                           cout<<arr[i]<<" ";
                  }
         }
   void setValue(int index, int value) {
      arr[index] = value;
   }

   int getValue(int index) const {
      return arr[index];

   }

   int getSize()
   {
         return arrSize;
   }

};
```

```
int main() {
   DynamicArray myArray(5);
   for (int i = 0; i < 5; i++) {
      myArray.setValue(i, i);
   }

         myArray.printArray();
   DynamicArray i=myArray;

         i.printArray();
   return 0;
}
```

Output:

```
0 1 2 3 4

Copy constructor called

0 1 2 3 4

Destructor called

Destructor called
```