Constructors

A **Constructor** is a member function of a class that initializes objects of a class. In C++, Constructor is automatically called when an object (instance of the class) is created. It is a special member function of the class.

How constructors are different from a normal member function?

A constructor differs from member-functions in the following ways:

- Constructor has the same name as the Class itself.
- Constructors don’t have a return type.
- A Constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Default Constructor

It is not mandatory for the programmer to write a constructor for each class. C++ by default provides a default constructors with no parameters, and no statements for the body. Much like being:

```
Employee() {}   //as per our example
```

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;
};

int main()
```

```
{
    Employee emp;
    return 0;
}
```

In the above code, once the object is created, the default constructor is called. We can overload the default constructor (covered below).

Constructor Overloading

We need to first get a grasp of overloading first, which is explained as:

Overloading: Having the same name for a member-function/constructor as long as the list of arguments is different is called overloading. In such a case, depending upon the arguments passed, the appropriate overloaded function is deduced and called. An example of constructor overloading:

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;

    Employee()
    {
        id = "";
        name = "";
        years = 0;
    }

    // Overloaded constructor
    Employee(string id, string name, int years)
    {
        this->id = id;
        this->name = name;
        this->years = years;
    }
```

```
    // Overloaded constructor
    Employee(string id, string name)
    {
        this->id = id;
        this->name = name;
        years = 0;
    }

    void getDetails()
    {
        cout << "ID: " << id << ", Name: " << name
             << ", Experience: " << years << endl;
    }
};

int main()
{
    // 1st constructor is called
    Employee emp1;

    // 2nd constructor is called
    Employee emp2("GFG123", "John", 4);

    // 3rd constructor is called
    // where years is 0 (no experience for a fresher)
    Employee fresher("GFG456", "James");

    emp1.getDetails();
    emp2.getDetails();
    fresher.getDetails();

    return 0;
}
```

Output:

```
ID:, Name:, Experience: 0
ID: GFG123, Name: John, Experience: 4
ID: GFG456, Name: James, Experience: 0
```

Member Initializer List

Member Initialization List is a new syntactic construct introduced in modern C++, which allows us to write concise initialization code in constructors. The basic syntax is as:

```
Constructor(< arguments >) : < mem1(arg1), mem2(arg2),
...., > {
    //additional code to execute after initialization
}
```

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;

    Employee(string id, string name, int years)
        : id(id), name(name), years(years)
    {
        // extra code to run after initialization
    }

    // does the same as:
    // Employee(string id, string name, int years) {
    // this->id = id;
    // this->name = name;
    // this->years = years;

    //     // extra-code to run after initialization
    // }

    void getDetails()
    {
        cout << "ID: " << id << ", Name: " << name
             << ", Experience: " << years << endl;
    }
```

```
};

int main()
{
    Employee emp("GFG123", "John", 4);

    emp.getDetails();

    return 0;
}
```

Output:

```
ID: GFG123, Name: John, Experience: 4
```

Constructor Chaining (Delegation)

Constructor chaining/delegation is the process of re-using constructors by others to avoid writing repeated code. This is done by calling one constructor to set common values by other constructors. As an example:

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public: // public access-modifier
    string id, name;
    int years;

    Employee(string id)
        : id(id)
    {
    }

    // uses constructor1                constructor1 call
    Employee(string id, string name)
        : Employee(id)
    {
```

```
        this->name = name;
    }

    // uses constructor2                        constructor2 c
    Employee(string id, string name, int years)
        : Employee(id, name)
    {
        this->years = years;
    }

    void getDetails()
    {
        cout << "ID: " << id << ", Name: " << name
             << ", Experience: " << years << endl;
    }
};

int main()
{
    Employee emp("GFG123", "John", 4);
    emp.getDetails();

    return 0;
}
```

Output:

```
ID: GFG123, Name: John, Experience: 4
```

Destructors

Destructors like constructors are special members of a class that is executed once the lifetime of the object expires. It is like the final clean-up code required before deleting the class instance. Unlike JAVA, C++ doesn't perform automatic garbage collection. Hence, it often becomes the responsibility of the developer to de-allocate memory (not required further). A classic example:

```
#include <bits/stdc++.h>
using namespace std;

char* p_chr;

class String {
public:
    char* s;
    int size;

    String(char* c)
    {
        size = strlen(c);
        s = new char[size + 1];
        p_chr = s; // assign to global variable
        strcpy(s, c);
    }
};

void func()
{
    String str("Hello World");
}

int main()
{
    func();
    cout << p_chr << endl;
    return 0;
}
```

Output:

```
Hello World
```

In the above code, we dynamically create a character array in our constructor, where we copy the string passed as an argument. Since it is a dynamically-allocated memory, once, the

lifetime of *str* object expires (inside the *func( )* call), still the memory is not de-allocated, as in main(), when we print *p_chr* , we get the string. Thus, we can see that proper clean-up of pointer references doesn't occur. Thus, we need destructors where we would explicitly de-allocate the memory and perform other clean-up code. Syntax of destructor:

```
~ Classname { //clean-up code }
```

The above code with destructors:

```
#include <bits/stdc++.h>
using namespace std;

char* p_chr;

class String {
public:
    char* s;
    int size;

    String(char* c)
    {
        size = strlen(c);
        s = new char[size + 1];
        p_chr = s;
        strcpy(s, c);
    }

    // Destructor
    ~String()
    {
        delete[] s;
    }
};

void func()
{
    String str("Hello World");
}
```

```
int main()
{
    func();
    cout << p_chr << endl;
    return 0;
}
```

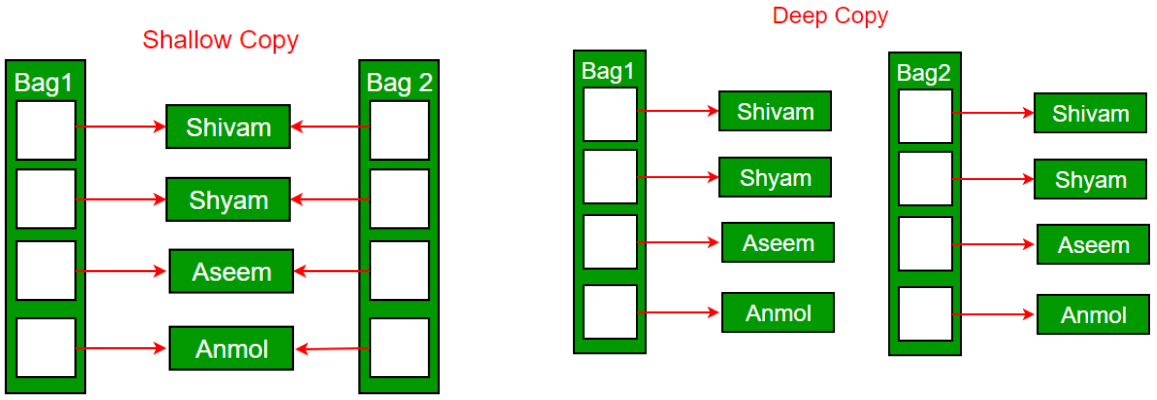
The above code prints nothing because *p_chr* reference is gone due to destructor call.

Copy Constructor

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

```
Classname (const Classname &object);
```

Copy constructor, in general, is not required to be defined by the user, as the compiler automatically provides a default copy constructor. However, this default copy constructor performs a shallow copy only (i.e. copy values only). This results in pointer variables pointing the same instances upon copy. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection. etc.



As an example:

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Array {
public:
    int n;
    int* ref;

    Array(int n)
        : n(n)
    {
        ref = new int[n];

        for (int i = 0; i < n; i++)
            *(ref + i) = i;
    }
};

int main()
{
    Array arr1(10);

    // copy constructor called
    // at this point
    Array arr2 = arr1;

    // changing n-value in 2nd instance
    arr2.n = 5;

    // changing array-values in 2nd instance
    for (int i = 0; i < 10; i++)
        *(arr2.ref + i) *= 2;

    cout << "n-value of 1st instance: " << arr1.n << endl;
    cout << "Array values of 1st instance:\n";
    for (int i = 0; i < 10; i++)
        cout << *(arr1.ref + i) << " ";
    cout << endl;
```

```
    return 0;
}
```

Output:

```
n-value of 1st instance: 10
Array values of 1st instance:
0 2 4 6 8 10 12 14 16 18
```

In the above code, we find the value of member *n* for *t1* not modified as it is not a pointer value. Thus, upon copying new instance of *n* got created for *t2* . Any change to n in *t2* didn't change *t1.n*. However, *t1.ref* is a pointer. So, upon copy-constructor call, the address value got copied to *t2.ref*, and thus, any change at t2.ref (as we here are multiplying by 2), gets reflected at t1.ref also because both of them are pointing to the same array. This is an example of a shallow-copy. To fix this, we write our custom copy-constructor:

```
#include <bits/stdc++.h>
using namespace std;

class Array {
public:
    int n;
    int* ref;

    Array(int n)
        : n(n)
    {
        ref = new int[n];

        for (int i = 0; i < n; i++)
            *(ref + i) = i;
    }

    // copy-contructor definition
    Array(const Array& obj)
        : n(obj.n)
    {
        ref = new int[n];

        for (int i = 0; i < n; i++)
```

```
            *(ref + i) = *(obj.ref + i);
    }
};

int main()
{
    Array arr1(10);

    // copy constructor called
    // at this point
    Array arr2 = arr1;

    // changing n-value in 2nd instance
    arr2.n = 5;

    // changing array-values in 2nd instance
    for (int i = 0; i < 10; i++)
        *(arr2.ref + i) *= 2;

    cout << "n-value of 1st instance: " << arr1.n << endl;
    cout << "Array values of 1st instance:\n";
    for (int i = 0; i < 10; i++)
        cout << *(arr1.ref + i) << " ";
    cout << endl;

    return 0;
}
```

Output:

```
n-value of 1st instance: 10
Array values of 1st instance:
0 1 2 3 4 5 6 7 8 9
```

We can see that in our copy-constructor, we re-create a dynamic memory for the array and then copy the values. This results in a deep-copy. Meaning changes in *arr2* doesn't affect *arr1*.