- **INHERITANCE CONSTRUCTOR**

How C++ Constructors are called in Inheritance with Examples

In this article, I am going to discuss How C++ Constructors are called in Inheritance with Examples. Constructor is a class member function with the same name as the class name. The main job of the constructor is to allocate memory for class objects. Constructor is automatically called when the object is created. It is very important to understand how constructors are called in inheritance.

How we can access Base class members using Derived Class Object in C++?

We know when we create an object, then automatically the constructor of the class is called and that constructor takes the responsibility to create and initialize the class members. Once we create the object, then we can access the data members and member functions of the class using the object.

In inheritance, we have Base/Parent/Superclass as well as Derived/Child/Subclass. If we create an object of the Base class, then the base class constructor is called and initializes the base class members, and then using the base class object, we can call the data members and member functions of the base class.

When we create an instance of the Derived class, then the Derived Class constructor is called and initializes the Derived Class members. But using the Derived class object we can access both the base class and derived class members**. How?**

How we can access the base class members using the derived class object? How the base class constructor is called? When the base class members are created?

Actually, when we create an object of the Derived class, the derived class constructor is called and initializes the derived class members. Also, the derived class constructor is either implicitly or explicitly called the Base class constructor and when the base class constructor is called, base class members are also created and initialized. This is the reason using the derived class object we can access both base class and derived class members.

How C++ Constructors are called in Inheritance?

Now let us see an example to understand how the C++ Constructors are called in inheritance. Here, we are taking a simple example. We are taking an example of class Base. In the base class we will not write anything, just write constructors as follows**:**

```
class Base
{
   public:
      Base ()
      {
         cout << "Default of Base" << endl;
      }
```

```
    Base (int x)
    {
       cout << "Param of Base " << x << endl;
    }
};
```

This is our Base class which has two constructors. The first constructor is the default constructor that will print “Default of Base” when the object is created. The second constructor is parameterized constructor that will print “Param of Base” then it will print the value of x.

Now we will write a class called Derived that will be inheriting from the Base class. Inside the Derived class, we will not write any things, just defined the constructors as follows.

```
class Derived : public Base
{
   public:
      Derived ()
      {
         cout << "Default of Derived" << endl;
      }
      Derived (int a)
      {
         cout << "Param of Derived : " << a << endl;
      }
};
```

This is our Derived class. This class is inherited from the Base class. It has two constructors. The first is a non-parameterized or default constructor that will print “Default of Derived” when called. The second is a parameterized constructor that will print “Param of Derived” then it will print the value of a.

So, we have two classes that are Base class with its default and parameterized constructor and the Derived class with its own default and parameterized constructor.

Now let us create an object of the Derived class and see how the constructors are executed. So, inside the main method, we will create an object of the Derived class as follows.

```
int main(){
    Derived d;
}
```

Here we have created an object d of class Derived and not passing any argument means which constructor we are calling? We are trying to call the Derived() constructor. But we also know that along with the Derived class constructor, the parent class i.e. Base class constructor will also execute. But there are two constructors in the parent class which constructor will execute? Default constructor i.e. Base(). So, by default, the default constructor of the parent class will be executed.

Example: Executing the Parent Class Default Constructor automatically in C++

```
#include <iostream>
using namespace std;
class Base
{
   public:
      Base ()
      {
         cout << "Default of Base" << endl;
      }
      Base (int x)
      {
         cout << "Param of Base " << x << endl;
      }
};
class Derived : public Base
{
   public:
      Derived ()
      {
         cout << "Default of Derived" << endl;
      }
      Derived (int a)
      {
         cout << "Param of Derived" << a << endl;
      }
};
int main()
{
   Derived d;
}
```

Output:

```
Default of Base
Default of Derived
```

Executing the Parent Class Default Constructor automatically in C++

Let us see what happened here. First, it displayed, “Default of Base” and then it displayed “Default of Derived”. That means when you create an object of the Derived class then first the base class constructor will be executed then the Derived class constructor will be executed.

So, the point that you need to remember is whenever you are creating an object of a derived class then first the constructor of the base class will be executed and then the constructor of the derived class will be executed.

Which constructor of the parent class will be executed?
Always the default constructor of the parent class will be executed. Let us pass some value in object d as follows.

```
int main(){
    Derived d (5);
}
```

Here we have passed 5 as a parameter in the constructor of the Derived class object. In this case, we have created an object of the Derived class by calling the parameterized constructor with a value of 5. But we know very well that the Derived class constructor will not execute first. The Base class constructor will execute. So which constructor will execute in the Base class? Again, the default constructor of the base will execute. So, first "Default of Base" will be printed on the screen. Then after that, it will come back and execute the Derived class parameterized constructor. "Param of Derived 5" will be printed.
The complete example is given below.

```
#include <iostream>
using namespace std;
class Base
{
   public:
      Base ()
      {
         cout << "Default of Base" << endl;
      }
      Base (int x)
      {
         cout << "Param of Base " << x << endl;
      }
};
class Derived : public Base
{
   public:
      Derived ()
      {
         cout << "Default of Derived" << endl;
      }
      Derived (int a)
      {
         cout << "Param of Derived : " << a << endl;
```

```
    }
};
int main()
{
   Derived d(5);
}
```

Output:

```
Default Of Base
Param of Derived
```

So still the default constructor of Base class and then parameterized constructor of Derived class has executed.

How to execute the Parameterized Constructor of Base class in Inheritance?
Now we want to call the parameterized constructor of the Base class when the object of the Derived classes is executed. So, for that, we should have a special constructor in the Derived class as follows which will call the base class parameterized constructor.

```
Derived(int x, int a) : Base(x){
    cout << "Param of Derived " << a;
}
```

Here we have written another parameterized constructor in the Derived class. This constructor is taking two integer type parameters that are x and a. Then we have written ": Base (x)". So, here we are calling the parameterized constructor of the Base class with x as a parameter. Next, we have written a printing statement "Param of Derived" and then print the value of a. So, here from the Derived class constructor we are explicitly calling the parameterized constructor of the Base class. So let us write another statement inside the main function as follows:

```
int main(){
    Derived d (25, 15);
}
```

Here we are giving two parameters in the constructor of the Derived object. Now, the parameterized constructor of the Derived class will be called which is taking two parameters.

This constructor will take 25 in x and 15 in a. Then the Derived class constructor will call Base(25). The Base class parameterized constructor will be called. So, in this way, we can call the base class parameterized constructor from the derived class constructor. The complete example code is given below.

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

```
{
   public:
      Base ()
      {
         cout << "Default of Base" << endl;
      }
      Base (int x)
      {
         cout << "Param of Base " << x << endl;
      }
};
class Derived : public Base
{
   public:
      Derived ()
      {
         cout << "Default of Derived" << endl;
      }
      Derived (int a)
      {
         cout << "Param of Derived : " << a << endl;
      }
      Derived(int x, int a) : Base(x)
      {
       cout << "Param of Derived " << a;
      }
};
int main()
{
   Derived d(25, 15);
}
```

Output:

```
Param of Base 25
Param of Derived 15
```

Multiple Inheritance Constructor

https://www.geeksforgeeks.org/constructor-in-multiple-inheritance-in-cpp/

Multilevel Inheritance Constructor

https://www.geeksforgeeks.org/constructor-in-multilevel-inheritance-in-cpp/

- **COMPOSITION VS INHERITANCE**

Composition	Inheritance
Composition is a ‘has-a’ relationship	Inheritance represents the ‘is-a’ relationship
We can achieve multiple inheritance using composition	Java doesn’t allow multiple inheritance
Composition does not create a hierarchy of classes	It creates a hierarchy of class
Composition does not allow direct access to the members of the composed objects	A child class can access all public and protected members of the parent class
The composition can be more flexible and allows objects to be reused in different contexts	Inheritance creates a tight coupling between the parent and child classes
Changes to the composed objects do not affect other composed objects	Changes to the parent class can affect all child classes
Composition allows code reuse even from final classes.	Inheritance cannot extend the final class

- **INITIALIZER LIST**

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon. Following is an example that uses the initializer list to initialize x and y of Point class.

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

class Point {
private:
    int x;
    int y;

public:
    Point(int i = 0, int j = 0): x(i), y(j) {}
    /* The above use of Initializer list is optional as the
        constructor can also be written as:
        Point(int i = 0, int j = 0) {
            x = i;
            y = j;
        }
    */

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

int main()
{
    Point t1(10, 15);
    cout << "x = " << t1.getX() << ", ";
    cout << "y = " << t1.getY();
    return 0;
}
```

Output:

x = 10, y = 15

The above code is just an example for syntax of the Initializer list. In the above code, x and y can also be easily initialized inside the constructor. But there are situations where initialization of data members inside constructor doesn't work and Initializer List must be used. The following are such cases:

1. For Initialization of Non-Static const Data Members

const data members must be initialized using Initializer List. In the following example, "t" is a const data member of Test class and is initialized using Initializer List. Reason for

initializing the const data member in the initializer list is because no memory is allocated separately for const data member, it is folded in the symbol table due to which we need to initialize it in the initializer list.

Also, it is a Parameterized constructor and we don't need to call the assignment operator which means we are avoiding one extra operation.
Example:

```
// C++ progmram to demonstrate the use of
// initializer list to initialize the const
// data member
#include<iostream>
using namespace std;

class Test {
    const int t;
public:
//Initializer list must be used
    Test(int t):t(t) {}
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}
```

Output:
10

2. For Initialization of Reference Members
Reference members must be initialized using the Initializer List. In the following example, "t" is a reference member of the Test class and is initialized using the Initializer List.

Example

```
// Initialization of reference data members
#include<iostream>
using namespace std;

class Test {
    int &t;
public:
    Test(int &t):t(t) {} //Initializer list must be used
    int getT() { return t; }
};

int main() {
```

```
    int x = 20;
    Test t1(x);
    cout<<t1.getT()<<endl;
    x = 30;
    cout<<t1.getT()<<endl;
    return 0;
}
```

Output:

20
30

3. For Initialization of Member Objects that do not have a Default Constructor

In the following example, an object “a” of class “A” is a data member of class “B”, and “A” doesn’t have a default constructor. Initializer List must be used to initialize “a”.

```
// C++ progmam to initialize a member object without default
// constructor
#include <iostream>
using namespace std;

class A {
    int i;

public:
    A(int);
};

A::A(int arg)
{
    i = arg;
    cout << "A's Constructor called: Value of i: " << i
         << endl;
}

// Class B contains object of A
class B {
    A a;

public:
    B(int);
};

B::B(int x) : a(x)
{ // Initializer list must be used
    cout << "B's Constructor called";
}
```

```
int main()
{
	B obj(10);
	return 0;
}
```
Output:
A's Constructor called: Value of i: 10
B's Constructor called

If class A had both default and parameterized constructors, then Initializer List is not a must if we want to initialize “a” using the default constructor, but it is must to initialize “a” using the parameterized constructor.

4. For Initialization of Base Class Members
Like point 3, the parameterized constructor of the base class can only be called using the Initializer List.

Example

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

class A {
	int i;
public:
	A(int );
};

A::A(int arg) {
	i = arg;
	cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B is derived from A
class B: A {
public:
	B(int );
};

B::B(int x):A(x) { //Initializer list must be used
	cout << "B's Constructor called";
}

int main() {
```

```
    B obj(10);
    return 0;
}
```
Output
A's Constructor called: Value of i: 10
B's Constructor called

5. When the Constructor's Parameter Name is the Same as Data Member

If the constructor's parameter name is the same as the data member name then the data member must be initialized either using this pointer or Initializer List. In the following example, both the member name and parameter name for A() is "i".

Example

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

class A {
    int i;

public:
    A(int);
    int getI() const { return i; }
};

A::A(int i) : i(i)
{
} // Either Initializer list or this pointer must be used
/* The above constructor can also be written as
A::A(int i) {
    this->i = i;
}
*/

int main()
{
    A a(10);
    cout << a.getI();
    return 0;
}
```
Output:
10

6. For Performance Reasons

It is better to initialize all class variables in the Initializer List instead of assigning values inside the body. Consider the following example:

Example

```
// Without Initializer List
class MyClass {
      Type variable;
public:
      MyClass(Type a) { // Assume that Type is an already
                                    // declared class and it has appropriate
                                    // constructors and operators
      variable = a;
      }
};
```

Here compiler follows following steps to create an object of type MyClass

1. Type's constructor is called first for "a".

2. Default construct "variable"

3. The assignment operator of "Type" is called inside body of MyClass() constructor to assign

variable = a;

4. And then finally destructor of "Type" is called for "a" since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

```
// With Initializer List
class MyClass {
   Type variable;
public:
   MyClass(Type a):variable(a) {   // Assume that Type is an already
                // declared class and it has appropriate
                // constructors and operators
   }
};
```

With the Initializer List, the following steps are followed by compiler:

1. Type's constructor is called first for "a".
2. Parameterized constructor of "Type" class is called to initialize: variable(a). The arguments in the initializer list are used to copy construct "variable" directly.
3. The destructor of "Type" is called for "a" since it goes out of scope.

As we can see from this example if we use assignment inside constructor body there are three function calls: constructor + destructor + one addition assignment operator call. And if we use Initializer List there are only two function calls: copy constructor + destructor call. See this post for a running example on this point.

This assignment penalty will be much more in “real” applications where there will be many such variables. Thanks to ptr for adding this point.

Parameter vs Uniform Initialization in C++

It is better to use an initialization list with uniform initialization {} rather than parameter initialization () to avoid the issue of narrowing conversions and unexpected behavior. It provides stricter type-checking during initialization and prevents potential narrowing conversions

Code using parameter initialization ()

```
#include <iostream>

class Base {
   char x;

public:
   Base(char a)
      : x{ a }
   {
   }

   void print() { std::cout << static_cast<int>(x); }
};

int main()
{
   Base b{ 300 }; // Using uniform initialization with {}
   b.print();
   return 0;
}
```

Output:

44

In the above code, the value 300 is out of the valid range for char, which may lead to undefined behavior and potentially incorrect results. The compiler might generate a warning or error for this situation, depending on the compilation settings.

Code using uniform initialization {}

By using uniform initialization with {} and initializing x with the provided value a, the compiler will perform stricter type-checking and issue a warning or error during compilation, indicating the narrowing conversion from int to char.

Here is code with uniform initialization {} , which results in a warning and thus better to use.

```
#include <iostream>
```

```
class Base {
   char x;

public:
   Base(char a)
      : x{ a }
   {
   }

   void print() { std::cout << static_cast<int>(x); }
};

int main()
{
   Base b{ 300 }; // Using uniform initialization with {}
   b.print();
   return 0;
}
```