

Translated from French to English - [www.onlinedoctranslator.com](http://www.onlinedoctranslator.com)

University of Monastir

## Course: C++ Programming

### Chapter 5: Inheritance in C++

Directed by:

Dr. Sakka Rouis Taoufik

<https://github.com/srtaoufik/coursCpp>

1

#### Chapter 5: Inheritance in C++

##### I. The Inheritance:

##### Definition

Inheritance (called class derivation) is the act of defining a new class (called a child class) based on the definition of an already existing class.

The new class then inherits the attributes and methods of the base (or parent) class in addition to its own members (its specific attributes and/or methods).

2

## Chapter 5: Inheritance in C++

### I. The inheritance: Derivation Syntax in C++

```
class DerivedClassName : [Derivation_Mode] BaseClassName {
```

// List of new members of the derived class

};

- **DerivedClassName**: designates the new class created by derivation.

- **BaseClassName**: designates the name of the class from which the derivation is made.

- **Derivation\_Mode**: is used to indicate the mode in which the derivation will be made. This mode determines the access rights that will be assigned to the members of the base class in the derived class. The modes that are defined in C++ are:*public,protectedAndprivate*.

**RK.** The accessibility of members of a base class from a derived class depends on:

- access rights of these members in the base class,

- of the derivation mode used.

The following table lists all possible scenarios regarding this accessibility:

3

## Chapter 5: Inheritance in C++

### I. The Inheritance: Derivation Syntax in C++

Mode de dérivation	Statut du membre dans la classe de base	Statut du membre dans la classe dérivée	Accessibilité du membre dans la classe dérivée	Accessibilité du membre par les utilisateurs de la classe dérivée
public	public	public	Accessible	Accessible
	protected	protected	Accessible	Inaccessible
	private	private	Inaccessible	Inaccessible
protected	public	protected	Accessible	Inaccessible
	protected	protected	Accessible	Inaccessible
	private	private	Inaccessible	Inaccessible
private	public	private	Accessible	Inaccessible
	protected	private	Accessible	Inaccessible
	private	private	Inaccessible	Inaccessible

**RK.** The default pass mode (if nothing is mentioned) is "*private*".

**RK.** Members of a base class that are accessible in the derived class are manipulated in the latter directly through their names in the manner of members of the derived class.

4

## Chapter 5: Inheritance in C++

### I. The Inheritance:

### Derivation Syntax in C++

#### Example

```
class Document {
    char Reference[10];
    char Title[100];
    void Display();
};

class Book: public Document {
char Author[10];
char Editor[100];
};
```

5

## Chapter 5: Inheritance in C++

### I. The Inheritance:

### Derivation Syntax in C++

<pre>class A { int a1; protected: int a2; public : int a3; }; class B : private A { private int b1, b2; protected: void f() {     b1 = a1*2; //Error because a1 is inaccessible in B } void g(){     b2 = a2*a3;     //OK because a2 and a3 are accessible in B } };</pre>	<pre>void main() { A Obj1; B Obj2; Obj1.a2 = 6; //Error because a2 is protected in A Obj2.a3 = 8; //Error because a3 is private in B Obj1.a3 = 8; //ok no problem }</pre>
--	---

6

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

Redefining methods is the act of proposing **a new definition** in the derived class **of an already existing method** in the base class.

- Generally the base method and the redefined one provide the same functionality.

However, the redefinition is often done in order to take into account the specificities of the derived class.

7

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

```
class A {
public:
    int att_A;
    A (int p) { att_A = p; }
    void Display(){ cout<<"The value of att_A is: "<<att_A<<endl; }
};

class B : public A {
public:
    int att_B;
    B(int p1, int p2) : A(p1){ att_B = p2; }
};

void main() {
    B b(5,9);
    b.Show();
}
```

The method *Display* called from *object b* will generate the following output:  
**The value of att\_A is: 5**

8

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

**New definition of B:**

```
class B : public A {
public :
    int att_B;
    B(int p1, int p2) : A(p1){ att_B = p2;}
    void Display() { //redefinition of the Display function
        cout<<"The value of att_A is: "<<att_A<<endl;
        cout<<"The value of att_B is: "<<att_B<<endl;
    }
};
```

With this definition of B the following code:

```
B b(5,9);
b.Show();
```

In this case, the method *Display* called from object *b* will generate the following output:

The value of att\_A is: 5  
 The value of att\_B is: 9

9

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

#### Calling members of a base class redefined in the derived class

- Members of a base class that are accessible in the derived class are manipulated in the latter directly through their names in the same way as members of the derived class. However, if a member (attribute or method) is redefined, it will be hidden in the derived class by the redefinition. To be able to call it anyway in the derived class, the name of this member must be associated with that of the base class using the scope resolution operator according to the following syntax:

**BaseClassName::MemberName;**

**RK.** If the member in question is public in the derived class then it can be manipulated from the objects of the latter according to the following syntax:

**ObjectName.BaseClassName::MemberName;**

10

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

**Noticed :**

In the previous example, it is more interesting to call the method *Display* of the class *A* in the definition of *Display* of *B* to ensure the display of *att\_A* (in order to avoid rewriting code that has already been written). However, the following definition of the method *Display* of *B*:

```
void Display() {
    Display();
    cout<<"The value of att_B is: "<<att_B<<endl;
}
```

will be interpreted as a recursive call of *Display* of *B*. To achieve the desired effect, redefining *Display* must be done as follows:

```
void Display() {
    A::Display ();
    cout<<"The value of att_B is: "<<att_B<<endl;
}
```

11

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

**RQ1. Difference between redefinition and overdefinition:**

Overdefinition should not be confused with redefinition. In redefinition, the base method and the redefined one must have exactly the same signature. This is not the case in overdefinition.

**RQ2. Compatibility between base class and derived class:**

Generally speaking in OOP, there is a certain compatibility between an object of a base class and an object of a derived class. This compatibility goes in the following direction (Derived object → Base object). It is interpreted as follows: *any object of a derived class can be considered an object of the base class*.

- ➔ Any object of a derived class has all the members of an object of the base class. It can therefore replace it where the object of the base class is requested (replace it syntactically but this does not guarantee that the result will always be obtained). satisfying).
- ➔ The reverse operation is not true. In other words, an object of a base class cannot replace an object of a derived class.

12

## Chapter 5: Inheritance in C++

### II. Redefinition of methods

**Example :** Given the two classes A And B defined as follows:

```
class A {
public:
    int att_A;
    void Show() {cout<<" att_A:"<<att_A<<endl}
};

class B : public A {
public:
    int att_B;
    void Display() {
        cout<<" att_A :"<<att_A<<endl;
        cout<<" att_B :"<<att_B<<endl;
    }
};

A a , *pa;
B b , *pb;
...
```

a=b ; // OK  
b=a ; //Error  
  
pa = &b; // OK  
pb = &a; // Error  
  
pb = (A\*) &a; // OK, casting  
//Explicit conversion  
pb->att\_B; // Error  
  
pa->att\_A; // OK  
pa->att\_B; // Error  
  
pa->Show();  
/\*This instruction will cause  
the function to be  
called *Display* defined  
in *HA* \*/

## Chapter 5: Inheritance in C++

### III. Construction of derived objects

- Instantiating an object of a derived class necessarily generates the instantiation of the part of its attributes coming from the base class. This instantiation process is carried out

**In C++ in four steps:**

- o Allocation of memory space for attributes,
- o Call the constructor of the derived class.
- o Call and execute the base class constructor (initialization of attributes from the base class).
- o Execution of the derived class constructor (initialization of derived attributes).

## Chapter 5: Inheritance in C++

### III. Construction of derived objects

**A Case where the derived class has a parameterized constructor:**

**A.1. Cases where the base class has a parameterized constructor:**

DerivedClass(Arguments):BaseClass(Arguments) { ... ... ... }

*Example :*

```
class A {
public:
int att_A;
A(int tp): att_A(p) {
cout<<"Builder of A";
cout <<endl;
}
void Display() {
cout<<"att_A:"<<att_A;
cout <<endl;
}
};
```

```
class B: public A {
public:
int att_B;
B(int p1,int p2):A(p1),att_B(p2) {
cout<<"Builder of B"<<endl;
}
void Display(){
cout<<"att_A:"<<att_A;
cout<<" and att_B :"<<att_B;
cout <<endl;
};
...
B b(5,7); // OK
```

15

## Chapter 5: Inheritance in C++

### III. Construction of derived objects

**A Case where the derived class has a parameterized constructor:**

**A.2. Cases where the base class has a non-parameterized constructor:**

This constructor can be implicit or explicit. In this case, the explicit call of the base class constructor in the derived class is not necessary.

*Example :*

```
class A {
public:
int att_A;
A() {
cout<<"Builder of A";
cout <<endl;
}
void Display() {
cout<<"att_A:"<<att_A;
cout <<endl;
}
};
```

```
class B: public A {
public:
int att_B;
B(int p2): Att_B(p2) {
cout<<"Builder of B"<<endl;
}
void Display(){
cout<<"att_A:"<<att_A;
cout<<" and att_B :"<<att_B;
cout <<endl;
};
...
B b(7); // OK
```

16

## Chapter 5: Inheritance in C++

### III. Construction of derived objects

#### B. Cases where the derived class does not have a parameterized constructor

##### B.1. If the base class has parameterized constructors

In this case **instantiation of the derived object will not be possible**. Indeed, the instantiation of this object will be done with the default constructor generated by the compiler. However, the latter cannot make an automatic call to the parameterized constructor of the base class because it will not know which arguments to pass to it. This situation generates a compilation error.

```
class A {
public :
int att_A;
A(int p) : att_A(p) {
cout<<"Constructor of A"<<endl;
}
};
```

```
class B : public A {
public :
int att_B;
};
...
B b; // Error
```

17

## Chapter 5: Inheritance in C++

### III. Construction of derived objects

#### B. Cases where the derived class does not have a parameterized constructor

##### B.2. If the base class has a default constructor (without parameters)

In this case, it is the latter which will be automatically called by the default constructor of the derived class.

```
class A {
public :
int att_A;
};

};
```

```
class B : public A {
public :
int att_B;
};

...
B b; //OK: call of the default
constructor of B which calls its
// turn to the default constructor of A.
```

18

## Chapter 5: Inheritance in C++

### IV. Destruction of Derived Objects

- Calling the destructor of an object of a derived class automatically calls the destructor of the base class.
- The order of execution of these destructors is inverse to the order of execution of the constructors: Execution of the destructor of the derived class then execution of the destructor of the base class
- In more detail, the release is done as follows:
  - o Call and execute the destructor of the derived class.
  - o Call and execute the base class destructor.
  - o Freeing the memory used by the attributes (This does not concern the dynamic memory spaces pointed to by any pointer type attributes but the spaces occupied by the attributes themselves).

19

## Chapter 5: Inheritance in C++

### IV. Destruction of Derived Objects

**Example :**

```
class A {
public :
.... ...
~A(){ cout<<"Destructor of A"<<endl;}
.... ...
};

class B : public A {
public :
.... ...
~B(){ cout<<"Destructor of B"<<endl;}
.... ...
};
```

The destruction of an object declared as follows:

`B obj(5,7);`

will generate the following output

.... ....

`Destroyer of B`

`Destroyer of A`

20



## Chapter 5: Inheritance in C++

### V. Polymorphism

- The word polymorphism comes from Greek and means being able to take several shapes.
- In object-oriented programming, polymorphism is a concept that manifests itself by the ability of a method to execute in different ways depending on the nature of the object that calls it (base object or derived object).
- Polymorphism is primarily concerned with objects linked by an inheritance relationship.