National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2023, Lab Manual 11

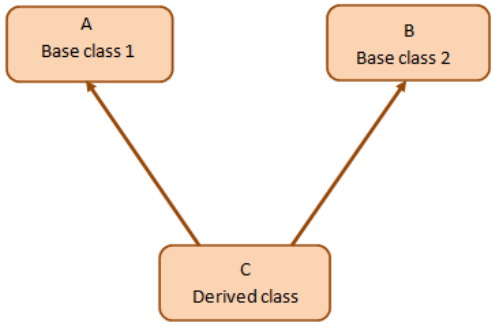
Course Code: CL1004	Course : Object Oriented Programming Lab
Instructor(s) :	SHAZIA PARAS SHAIKH

Contents

- Overview of multiple inheritance
- Diamond Problem
- Solving diamond problem using virtual keyword

Multiple Inheritance in C++

Multiple Inheritance is a feature of Object-Oriented Programming (OOP) where a subclass can inherit from more than one superclass. In other words, a child class can have more than one parent.
The figure below shows a pictorial representation of multiple inheritances.



In the above diagram, **class C** has **class A** and **class B** as its parents.
If we consider a real-life scenario, a child inherits from its father and mother. So a Child can be represented as a derived class with “Father” and “Mother” as its parents. Similarly, we can have many such real-life examples of multiple inheritance.
In multiple inheritance, the constructors of an inherited class are executed in the order that they are inherited. On the other hand, destructors are executed in the reverse order of their inheritance.

Now let's illustrate the multiple inheritance and verify the order of construction and destruction of objects.
Following code shows multiple inheritance

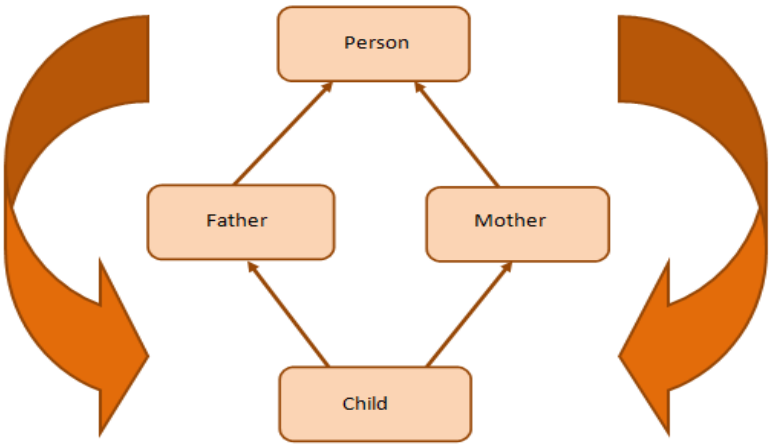
```
#include<iostream>
using namespace std;
class A //base class A with constructor and destructor
{
public:
    A() { cout << "class A::Constructor" << endl; }
    ~A() { cout << "class A::Destructor" << endl; }
};
class B //base class B with constructor and destructor
{
public:
    B() { cout << "class B::Constructor" << endl; }
    ~B() { cout << "class B::Destructor" << endl; }
};
class C : public B, public A //derived class C inherits class A and then class B (note the order)
{
public:
    C() { cout << "class C::Constructor" << endl; }
    ~C() { cout << "class C::Destructor" << endl; }
};
int main() {
    C c;
    return 0;
}
```

The output we obtain from the above program is as follows:

```
ClassB::Constructor
classA::Constructor
classC::Constructor
classC::Destructor
classA::Destructor
class B::Destructor
```

Now if we check the output, we see that the constructors are called in order B, A, and C while the destructors are in the reverse order.

The Diamond Problem



Here, we have a class **Child** inheriting from classes **Father** and **Mother**. These two classes, in turn, inherit the class **Person** because both Father and Mother are Person.
As shown in the figure, class Child inherits the traits of class Person twice—once from Father and again from Mother. This gives rise to ambiguity since the compiler fails to understand which way to go.
This scenario gives rise to a diamond-shaped inheritance graph and is famously called "The Diamond Problem."

Following code shows the diamond problem

```
#include<iostream>
using namespace std;
class Person { //class Person
public:
    Person(int x) { cout << "Person::Person(int) called" << endl; }
};

class Father : public Person { //class Father inherits Person
public:
    Father(int x) :Person(x) {
            cout << "Father::Father(int) called" << endl;
    }
};
```

```
class Mother : public Person { //class Mother inherits Person
public:
    Mother(int x) :Person(x) {
            cout << "Mother::Mother(int) called" << endl;
    }
};

class Child : public Father, public Mother { //Child inherits Father and Mother
public:
    Child(int x) :Mother(x), Father(x) {
            cout << "Child::Child(int) called" << endl;
    }
};

int main() {
    Child child(30);
}
```

Note that the com[piler would not give any error when the diamond problem occurs. It just creates ambiguity which can be seen int eh output of the above code.

```
Person::Person(int)called
Father::Father(int)called
Person::Person(int)called
Mother::Mother(int)called
Child::Child(int) called
```

Now you can see the ambiguity here. The Person class constructor is called twice: once when the Father class object is created and next when the Mother class object is created. The properties of the Person class are inherited twice, giving rise to ambiguity.
Since the Person class constructor is called twice, the destructor will also be called twice when the Child class object is destructed.

Fixing the Diamond Problem

The solution to the diamond problem is to use the **virtual** keyword. We make the two parent classes (who inherit from the same grandparent class) into virtual classes in order to avoid two copies of the grandparent class in the child class.

Let's change the above code and check the output:

```
#include<iostream>
using namespace std;
class Person { //class Person
public:
    Person() { cout << "Person::Person() called" << endl; } //Base constructor
    Person(int x) { cout << "Person::Person(int) called" << endl; }
```

```
};

class Father : virtual public Person { //class Father inherits Person
public:
    Father(int x) :Person(x) {
            cout << "Father::Father(int) called" << endl;
    }
};

class Mother : virtual public Person { //class Mother inherits Person
public:
    Mother(int x) :Person(x) {
            cout << "Mother::Mother(int) called" << endl;
    }
};

class Child : public Father, public Mother { //class Child inherits Father and Mother
public:
    Child(int x) :Mother(x), Father(x) {
            cout << "Child::Child(int) called" << endl;
    }
};

int main() {
    Child child(30);
}
```

Here we have used the **virtual** keyword when classes Father and Mother inherit the Person class. This is usually called “virtual inheritance," which guarantees that only a single instance of the inherited class (in this case, the Person class) is passed on.
In other words, the Child class will have a single instance of the Person class, shared by both the Father and Mother classes. By having a single instance of the Person class, the ambiguity is resolved.

The output of the above code is given below:

```
Person::Person()called
Father::Father(int)called
Mother::Mother(int)called
Child::Child(int) called
```

Here you can see that the class Person constructor is called only once.
One thing to note about virtual inheritance is that even if the parameterized constructor of the Person class is explicitly called by Father and Mother class constructors through initialization lists, **only the base constructor of the Person class will be called**.

This is because there's only a single instance of a virtual base class that's shared by multiple classes that inherit from it.
To prevent the base constructor from running multiple times, the constructor for a virtual base class is not called by the class inheriting from it. Instead, the constructor is called by the constructor of the concrete class.
In the example above, the class Child directly calls the base constructor for the class Person.