# Multiple Inheritance in C++

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.  The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B’s constructor is called before A’s constructor.

A class can be derived from more than one base class.

Eg:

(i) A CHILD class is derived from FATHER and MOTHER class  
(ii) A PETROL class is derived from LIQUID and FUEL class.

Syntax:

class A

{

... .. ...

};

class B

{

... .. ...

};

class C: public A,public B

{

... ... ...

};

* CPP

|  |
| --- |
| #include<iostream>  using namespace std;    class A  {  public:    A()  { cout << "A's constructor called" << endl; }  };    class B  {  public:    B()  { cout << "B's constructor called" << endl; }  };    class C: public B, public A  // Note the order  {  public:    C()  { cout << "C's constructor called" << endl; }  };    int main()  {      C c;      return 0;  } |

Output:

B's constructor called

A's constructor called

C's constructor called

The destructors are called in reverse order of constructors.

**The diamond problem** The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.  For example, consider the following program.

* CPP

|  |
| --- |
| #include<iostream>  using namespace std;  class Person {     // Data members of person  public:      Person(int x)  { cout << "Person::Person(int ) called" << endl;   }  };    class Faculty : public Person {     // data members of Faculty  public:      Faculty(int x):Person(x)   {         cout<<"Faculty::Faculty(int ) called"<< endl;      }  };    class Student : public Person {     // data members of Student  public:      Student(int x):Person(x) {          cout<<"Student::Student(int ) called"<< endl;      }  };    class TA : public Faculty, public Student  {  public:      TA(int x):Student(x), Faculty(x)   {          cout<<"TA::TA(int ) called"<< endl;      }  };    int main()  {      TA ta1(30);  } |

Person::Person(int ) called

Faculty::Faculty(int ) called

Person::Person(int ) called

Student::Student(int ) called

TA::TA(int ) called

In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed. So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities. *The solution to this problem is ‘virtual’ keyword*. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class. For example, consider the following program.

* CPP

|  |
| --- |
| #include<iostream>  using namespace std;  class Person {  public:      Person(int x)  { cout << "Person::Person(int ) called" << endl;   }      Person()     { cout << "Person::Person() called" << endl;   }  };    class Faculty : virtual public Person {  public:      Faculty(int x):Person(x)   {         cout<<"Faculty::Faculty(int ) called"<< endl;      }  };    class Student : virtual public Person {  public:      Student(int x):Person(x) {          cout<<"Student::Student(int ) called"<< endl;      }  };    class TA : public Faculty, public Student  {  public:      TA(int x):Student(x), Faculty(x)   {          cout<<"TA::TA(int ) called"<< endl;      }  };    int main()  {      TA ta1(30);  } |

Output:

Person::Person() called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

In the above program, constructor of ‘Person’ is called once. One important thing to note in the above output is, *the default constructor of ‘Person’ is called*. When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor. **How to call the parameterized constructor of the ‘Person’ class?**The constructor has to be called in ‘TA’ class. For example, see the following program.

* CPP

|  |
| --- |
| #include<iostream>  using namespace std;  class Person {  public:      Person(int x)  { cout << "Person::Person(int ) called" << endl;   }      Person()     { cout << "Person::Person() called" << endl;   }  };    class Faculty : virtual public Person {  public:      Faculty(int x):Person(x)   {         cout<<"Faculty::Faculty(int ) called"<< endl;      }  };    class Student : virtual public Person {  public:      Student(int x):Person(x) {          cout<<"Student::Student(int ) called"<< endl;      }  };    class TA : public Faculty, public Student  {  public:      TA(int x):Student(x), Faculty(x), Person(x)   {          cout<<"TA::TA(int ) called"<< endl;      }  };    int main()  {      TA ta1(30);  } |

Output:

Person::Person(int ) called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

In general, it is not allowed to call the grandparent’s constructor directly, it has to be called through parent class. It is allowed only when ‘virtual’ keyword is used. As an exercise, predict the output of following programs. **Question 1**

* CPP

|  |
| --- |
| #include<iostream>  using namespace std;    class A  {    int x;  public:    void setX(int i) {x = i;}    void print() { cout << x; }  };    class B:  public A  {  public:    B()  { setX(10); }  };    class C:  public A  {  public:    C()  { setX(20); }  };    class D: public B, public C {  };    int main()  {      D d;      d.print();      return 0;  } |

**Question 2**

* CPP

|  |
| --- |
| #include<iostream>  using namespace std;    class A  {    int x;  public:    A(int i) { x = i; }    void print() { cout << x; }  };    class B: virtual public A  {  public:    B():A(10) {  }  };    class C:  virtual public A  {  public:    C():A(10) {  }  };    class D: public B, public C {  };    int main()  {      D d;      d.print();      return 0;  } |