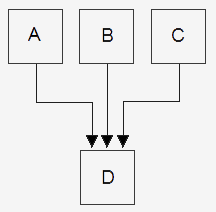
**Multiple Inheritance in C++**

# Multiple Inheritance

A class can be derived from more than one base class. This is called multiple inheritance.



# Constructors in Multiple Inheritance

As long as no base class constructor takes any arguments, the derived class need not have a constructor.

If any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

**In case of multiple inheritance**: base classes are constructed in the order in which they appear in the declaration of the derived class.

Constructors for virtual base classes are invoked before any non-virtual base classes.

If there are multiple virtual base classes, they are invoked in the order in which they are declared.

|  |  |
| --- | --- |
| Method of Inheritance | Order of execution |
| class D: public B {  }; | B(): Base constructor  D(): Derived constructor |
| class D: public B1, public B2 {  }; | B1(): base first  B2(): base second  D(): derived |
| class D: public B1, virtual public B2 {  }; | B2(): virtual base  B1(): ordinary base  D(): derived |

**Destructors in C++ are called in the opposite order of that of Constructors.**

Example:

#include<iostream>

using namespace std;

class A {

public:

A() { cout << "A's constructor called" << endl; }

~A() { cout << "A's destructor called" << endl; }

};

class B {

public:

B() { cout << "B's constructor called" << endl; }

~B() { cout << "B's destructor called" << endl; }

};

class C: public B, public A { // Note the order

public:

C() { cout << "C's constructor called" << endl; }

~C() { cout << "C's destructor called" << endl; }

};

class D: public B, virtual public A { // Note the order and virtual

public:

D() { cout << "D's constructor called" << endl; }

~D() { cout << "D's destructor called" << endl; }

};

int main() {

C c;

cout << endl;

D d;

cout << endl;

return 0;

}

Output:

B's constructor called

A's constructor called

C's constructor called

A's constructor called

B's constructor called

D's constructor called

D's destructor called

B's destructor called

A's destructor called

C's destructor called

A's destructor called

B's destructor called

# Ambiguity in Multiple Inheritance

## Base classes have common function

Two base classes have functions with the same name, while a class derived from both base classes has no function with this name. How do objects of the derived class access the correct base class function?

**Solution**: The problem is resolved using the scope-resolution operator to specify the class in which the function lies.

#include <iostream>

using namespace std;

class A {

public:

void show() { cout << "Class A" << endl; }

};

class B {

public:

void show() { cout << "Class B" << endl; }

};

class C : public A, public B

{

};

int main() {

C objC;

//objC.show(); // CE: request for member 'show' is ambiguous

objC.A::show();

objC.B::show();

return 0;

}

Output:

Class A

Class B

## The diamond problem

If you derive a class from two classes that are each derived from the same class. This creates a diamond-shaped inheritance tree.

**Solution**: ‘virtual’ keyword. We make the parent classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class.

#include<iostream>

using namespace std;

class Person {

public:

Person(int x) { cout << "Person::Person(int ) called" << endl; }

void display() {cout << "Person display" << endl; }

};

class Faculty : public Person {

public:

Faculty(int x):Person(x) {

cout<<"Faculty::Faculty(int ) called"<< endl;

}

};

class Student : 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) { // CW: base 'Student' will be initialized after [-Wreorder]

cout<<"TA::TA(int ) called"<< endl;

}

};

int main() {

TA ta1(30);

//ta1.display(); // CE: request for member 'display' is ambiguous

return 0;

}

Output:

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.

#include<iostream>

using namespace std;

class Person {

public:

Person() { cout << "Person::Person() called" << endl; }

Person(int x) { cout << "Person::Person(int ) called" << endl; }

void display() {cout << "Person display" << 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) { // CW: base 'Student' will be initialized after [-Wreorder]

cout<<"TA::TA(int ) called"<< endl;

}

};

int main() {

TA ta1(30);

ta1.display();

return 0;

}

Output:

Person::Person() called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

Person display

**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.

Solution for this problem is calling grandparent paramitirized constructor in derived class. See section [How to call the parameterized constructor of the grandparent (‘Person’) class in Multiple Inheritance?](#_How_to_call)

# How to call the parameterized constructor of the grandparent (‘Person’) class in Multiple Inheritance?

The constructor needs to be called in derived ('TA') class.

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.

#include<iostream>

using namespace std;

class Person {

public:

Person() { cout << "Person::Person() called" << endl; }

Person(int x) { cout << "Person::Person(int ) called" << endl; }

void display() {cout << "Person display" << 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) { // CW: base 'Faculty' will be initialized after

cout<<"TA::TA(int ) called"<< endl;

}

};

int main() {

TA ta1(30);

ta1.display();

return 0;

}

Output:

Person::Person(int ) called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

Person display

# Virtual Functions and Runtime Polymorphism in C++

See topic polymorphism

# Upcasting

In case of multiple inheritance an object can have more than one this pointer because the object represents more than one type during upcasting.

The following example demonstrates this point:

#include <iostream>

using namespace std;

class Base1 {

char c[0x10];

public:

void printthis1() {

cout << "Base1 this = " << this << endl;

}

};

class Base2 {

char c[0x10];

public:

void printthis2() {

cout << "Base2 this = " << this << endl;

}

};

class Member1 {

char c[0x10];

public:

void printthism1() {

cout << "Member1 this = " << this << endl;

}

};

class Member2 {

char c[0x10];

public:

void printthism2() {

cout << "Member2 this = " << this << endl;

}

};

class MI : public Base1, public Base2 {

Member1 m1;

Member2 m2;

public:

void printthis() {

cout << "MI this = " << this << endl;

printthis1();

printthis2();

m1.printthism1();

m2.printthism2();

}

};

int main() {

MI mi;

cout << "sizeof(mi) = " << hex << sizeof(mi) << " hex" << endl;

mi.printthis();

// A second demonstration:

Base1\* b1 = &mi; // Upcast

Base2\* b2 = &mi; // Upcast

cout << endl;

cout << "Base 1 pointer = " << b1 << endl;

cout << "Base 2 pointer = " << b2 << endl;

}

Output:

sizeof(mi) = 40 hex

MI this = 0x7fff39827ac0

Base1 this = 0x7fff39827ac0

Base2 this = 0x7fff39827ad0

Member1 this = 0x7fff39827ae0

Member2 this = 0x7fff39827af0

Base 1 pointer = 0x7fff39827ac0

Base 2 pointer = 0x7fff39827ad0

The starting address of the object corresponds to the address of the first class in the base-class list. Then the second inherited class is placed, followed by the member objects in order of declaration.

When the upcast to the Base1 and Base2 pointers occur, you can see that, even though they’re ostensibly pointing to the same object, they must actually have different this pointers, so the proper starting address can be passed to the member functions of each subobject.

The only way things can work correctly is if this implicit upcasting takes place when you call a member function for a multiply inherited subobject.