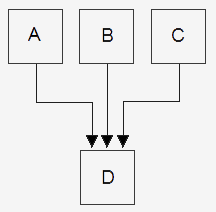
**Multiple Inheritance in C++**

# 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.

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

|  |  |
| --- | --- |
| 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 |

#include <iostream>

using namespace std;

class Base\_A {

public:

Base\_A() { cout << "Base\_A's constructor\n"; }

~Base\_A() { cout << "Base\_A's destructor\n"; }

};

class Base\_B {

public:

Base\_B() { cout << "Base\_B's constructor\n"; }

~Base\_B() { cout << "Base\_B's destructor\n"; }

};

class Derived\_A: public Base\_B, public Base\_A { // Note the order

public:

Derived\_A() { cout << "Derived\_A's constructor\n"; }

~Derived\_A() { cout << "Derived\_A's destructor\n"; }

};

class Derived\_B: public Base\_B, virtual public Base\_A { // Note the virtual base class

public:

Derived\_B() { cout << "Derived\_B's constructor\n"; }

~Derived\_B() { cout << "Derived\_B's destructor\n"; }

};

int main() {

Derived\_A da\_obj;

cout << '\n';

Derived\_B db\_obj;

cout << '\n';

return 0;

}

Output:

Base\_B's constructor

Base\_A's constructor

Derived\_A's constructor

Base\_A's constructor

Base\_B's constructor

Derived\_B's constructor

Derived\_B's destructor

Base\_B's destructor

Base\_A's destructor

Derived\_A's destructor

Base\_A's destructor

Base\_B's destructor

# 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 Base\_A {

public:

void display() { cout << "Base\_A\n"; }

};

class Base\_B {

public:

void display() { cout << "Base\_B\n"; }

};

class Derived : public Base\_B, public Base\_A {};

int main() {

Derived dobj;

// dobj.display(); // error: request for member 'display' is ambiguous

dobj.Base\_A::display();

dobj.Base\_B::display();

return 0;

}

Output:

Base\_A

Base\_B

## The diamond problem

### Problem

If you derive a class from two classes that are each derived from the same class. This creates a diamond-shaped inheritance tree. This causes problem of duplicate members and ambiguity when trying to access them.

#include <iostream>

using namespace std;

class Person {

public:

Person(int x) { cout << "Person constrcutor\n"; }

void display() { cout << "Person display\n"; }

};

class Faculty : public Person {

public:

Faculty(int x) : Person(x) { cout << "Faculty constrcutor\n"; }

};

class Student : public Person {

public:

Student(int x) : Person(x) { cout << "Student constrcutor\n"; }

};

class TA : public Faculty, public Student {

public:

TA(int x) : Student(x), Faculty(x) { // CW: base 'Student' will be initialized after [-Wreorder]

cout << "TA constrcutor\n";

}

};

int main() {

TA ta1(30);

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

return 0;

}

Output:

Person constrcutor

Faculty constrcutor

Person constrcutor

Student constrcutor

TA constrcutor

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.

### 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 constrcutor\n"; }

void display() { cout << "Person display\n"; }

};

class Faculty : virtual public Person {

public:

Faculty(int x) : Person(x) { cout << "Faculty constrcutor\n"; }

};

class Student : virtual public Person {

public:

Student(int x) : Person(x) { cout << "Student constrcutor\n"; }

};

class TA : public Faculty, public Student {

public:

// // Person class parameterized constrctor is called from here otherwise

// default constructor of Person will be called

TA(int x) : Person(x), Student(x), Faculty(x) {

cout << "TA constrcutor\n";

}

};

int main() {

TA ta1(30);

ta1.display();

return 0;

}

Output:

Person constrcutor

Faculty constrcutor

Student constrcutor

TA constrcutor

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.

Check this example [Solution](#_Solution)

# Virtual Functions and Runtime Polymorphism in C++

Check <06_CPP_OOP_Polymorphism_and_Virtual_Functions.docx>

# 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[5];

public:

void printThis\_base1() { cout << "Base1 this = " << this << '\n'; }

};

class Base2 {

char c[5];

public:

void printThis\_base2() { cout << "Base2 this = " << this << '\n'; }

};

class Member1 {

char c[5];

public:

void printThis\_m1() { cout << "Member1 this = " << this << '\n'; }

};

class Member2 {

char c[5];

public:

void printThis\_m2() { cout << "Member2 this = " << this << '\n'; }

};

class Derived : public Base1, public Base2 {

Member1 m1;

Member2 m2;

public:

void printThis() {

cout << "Derived this = " << this << '\n';

printThis\_base1();

printThis\_base2();

m1.printThis\_m1();

m2.printThis\_m2();

}

};

int main() {

Derived dobj;

cout << "sizeof(Derived) = " << sizeof(Derived) << '\n';

dobj.printThis();

// A second demonstration:

Base1\* b1 = &dobj; // Upcast

Base2\* b2 = &dobj; // Upcast

cout << '\n';

cout << "Base 1 pointer = " << b1 << '\n';

cout << "Base 2 pointer = " << b2 << '\n';

}

Output:

sizeof(Derived) = 20

Derived this = 0x7ffc7be32d60

Base1 this = 0x7ffc7be32d60

Base2 this = 0x7ffc7be32d65

Member1 this = 0x7ffc7be32d6a

Member2 this = 0x7ffc7be32d6f

Base 1 pointer = 0x7ffc7be32d60

Base 2 pointer = 0x7ffc7be32d65

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 sub object.

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

# References

<https://www.geeksforgeeks.org/c-plus-plus/#Inheritance>

Chapter 17 Inheritance | <https://www.learncpp.com/>

Thinking in C++, Volume 1, 2nd Edition