How virtual function works?

class A

{

//without virtual function

int x;

};

class B

{

//with virtual function

int b;

};

int main()

{

A a;

B b;

cout<<"size of object a is "<<sizeof(a);

cout<<"size of object a is "<<sizeof(b);

}

class B contains a virtual function whereas class A doesnot contain any virtual function. If the size of the objects of both the classes A and B is compared, then size of object b is 8 bytes more than the size of object a.The reason is, for every class which contains at least one virtual function, the compiler implicitly adds a virtual pointer.

=======================================================================================

If a class contains a virtual function then the compiler itself does two things.

1)If an object of that class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

2)Irrespective of whether the object is created or not, the class contains as a member a static array of function pointers called VTABLE. Cells of this table store the address of each virtual function contained in that class.

Virtual Functions in C++

The most common use of runtime binding is with base class pointers or references calling derived class functions. This is called runtime polymorphism and is implemented in C++ using virtual functions.

A virtual function is a member function that is declared within a base class with the virtual keyword and is re-defined (Overridden) by a derived class. When a class contains a virtual function, it can be overridden in its derived class and tells the compiler to perform dynamic linkage (or late binding) on the function. Now, the virtual functions are implemented in C++ by using vTable and vPtr.

What is VTable ?

==>VTable is a class specific. It is a static member of the class since all the objects of a class refer to the same virtual table. It co ntains the addresses of the respective virtual functions of that class. VTable is a constant array of function pointers which contains pointers to all virtual functions of the class. All the object of a class refer to its own virtual table.

==>If in the derived class,a function that is declared as virtual in the base class is not overridden, then the compiler uses the address of the base class version in the derived class virtual table.

What is Vptr (Virtual Pointer)?

==>Every object of a polymorphic class has a hidden pointer called virtual pointer(vptr). it is a pointer to VTable and is automatically initialized to the starting address of the VTable in the constructor.

==>The virtual pointer or \_vptr is a hidden pointer that is added by the compiler as a member of the class to point to the VTable of that class. Every object of a class containing virtual functions, a vptr is added to point to the vTable of that class. It’s important to note that vptr is created only if a class has or inherits a virtual function.

==>The setting up of the VTable for each polymorphic class, initializing the Vptr and inserting the codefor the function call happens automatically, the programmer doesnot have to worry about it.

consider Employee and Salesperson are polymorphic class.

==>when a class containing a virtual function is created or a class is derived from a class that contains virtual functions,compiler creates a unique VTable for that class.In VTable ,the compiler stores the addresses of all virtual functions.Each object contains vptr that points to VTable of that class.

==>The compiler places Vptr as a data member in the class created per object.Vptr is initialized to the starting address of the appropriate VTable.when salesperson object is created,it contains VPtr as one ofits data members which points to the appropriate VTable.

int main()

{

Employee \* ptr;

ptr=new Salesperson();

ptr->display();

}

The compiler picks up the Vptr of the object pointed by ptr.Thus instead of calling Employee::display() the compiler generates the code that says the function at Vptr should be called.The fectching of Vptr and the determination of the actual function address occur at runtime,thus late binding is achieved.

// C++ program to show the working of vtable and vptr

#include <iostream>

using namespace std;

// base class

class Base {

public:

virtual void function1()

{

cout << "Base function1()" << endl;

}

virtual void function2()

{

cout << "Base function2()" << endl;

}

virtual void function3()

{

cout << "Base function3()" << endl;

}

};

// class derived from Base

class Derived1 : public Base {

public:

// overriding function1()

void function1()

{

cout << "Derived1 function1()" << endl;

}

// not overriding function2() and function3()

};

// class derived from Derived1

class Derived2 : public Derived1 {

public:

// again overriding function2()

void function2()

{

cout << "Derived2 function2()" << endl;

}

// not overriding function1() and function3()

};

// driver code

int main()

{

// defining base class pointers

Base\* ptr1 = new Base();

Base\* ptr2 = new Derived1();

Base\* ptr3 = new Derived2();

// calling all functions

ptr1->function1();

ptr1->function2();

ptr1->function3();

ptr2->function1();

ptr2->function2();

ptr2->function3();

ptr3->function1();

ptr3->function2();

ptr3->function3();

// deleting objects

delete ptr1;

delete ptr2;

delete ptr3;

return 0;

}

Output

Base function1()

Base function2()

Base function3()

Derived1 function1()

Base function2()

Base function3()

Derived1 function1()

Derived2 function2()

Base function3()

==========================================================================

1)A class contains atleast one virtual function,that class is calles as polymorphic class

2)compiler implicitely add a virtual pointer for every polymorphic class i.e vptr

3)vptr is created per object

4)vtable is called as static array of function pointer.

5)vtable is created per class

6)vtable will store the address of all the virtual functions

7)vptr will point to the starting address of the table.

===========================================================================================

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.

==>The list of base classes memtioned in the derived class declaration must be seperated by commas and each of the base classes should use its own specifier . If the class specifier is not specified,default is private.

==>The order in which base class constructors are invoked is the same as the order in which they appear in the derived class constructor's member initialization list.

Syntax:

class A

{

... .. ...

};

class B

{

... .. ...

};

class C: public A,public B

{

... ... ...

};

#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.

==================================================================

#include<iostream>

using namespace std;

class A

{

int a;

public:

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

A(int p)

{

cout << "A's para constructor called" << endl;

}

};

class B

{

int b;

public:

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

B(int q)

{

cout << "B's para constructor called" << endl;

}

};

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

{

public:

C(int p,int q):A(p),B()

{

cout << "C's paraconstructor called" << endl;

}

};

int main()

{

C c(10,20);

return 0;

}

====================================================

Ambiguity Problem in Multiple Inheritance

In Multiple Inheritance, when a single class is derived from two or more base or parent classes. So, it might be possible that both the parent class have the same-named member functions, and it shows ambiguity when the child class object invokes one of the same-named member functions. Hence, we can say, the C++ compiler is confused in selecting the member function of a class for the execution of a program.

ex:

Base classes can contain a function with the same name,signature and returntype.

class A

{

... .. ...

};

class C

{

... .. ...

};

class B: public A,public B

{

... ... ...

};

B bobj

bobj.func();

==>Here,the function call is ambiguous and doesnot compile,because it is not clear whether it refers to A::func() or C::func().This ambiguity can be resolved by any one of the following 2 ways.

1)Making the call explicit by resolving the function call with the classname i.e specifying either bobj.A::func() or bobj.c::func()

2)overriding the func() in class B . then the statement bobj.func() will call func() of class B and the call will not be ambiguous.

#include <iostream>

#include <conio.h>

using namespace std;

// create class A

class A

{

public:

void show()

{

cout << " It is the member function of class A " << endl;

}

};

// create class B

class B

{

public:

void show()

{

cout << " It is the member function of class B " << endl;

}

};

// create a child class to inherit the member function of class A and class B

class child: public A, public B

{

public:

void disp()

{

cout << " It is the member function of the child class " << endl;

}

};

int main ()

{

// create an object of the child class to access the member function

child ch;

ch.show(); // It causes ambiguity

ch.disp();

return 0;

}

==================================================================

Diamond/Hybrid Inheritance:

In diamond inheritance,one child has two parents and both the parents share a common parent. When in a hierarchy, a class is derived from two classes and both the classes are derived from a common base class,it is termed as diamond inheritance.

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.

#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.

=========================================================================================

#include<iostream>

using namespace std;

class A

{

public:

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

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

};

class B:virtual public A

{

public:

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

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

};

class C:virtual public A // Note the order

{

public:

C()

{

cout << "C's default constructor called" << endl;

}

C(int a)

{

cout << "C's para constructor called" << endl;

}

};

class D:public C,public B // Note the order

{

public:

D(int p,int q,int r):C( q),B(),A(p)

{

cout << "D's para constructor called" << endl;

}

};

int main()

{

D dobj(10,20,30);

}

===========================================================================

=

#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 {0

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.

#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.