**Virtual**:

**What?**

Virtual is always related to inheritance and polymorphism. With virtual method or function the behaviour can be overridden within an inheriting class by function with same signature. Virtual function is generally used for customization of the derived class implementation.

A virtual method implementation is determined at runtime based on the actual type of the invoking object.

**Why?**

There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.

**When?**

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

**How?**

Late binding (runtime) is done in accordance to the contents of the pointer.

Early binding (compile time) is done in accordance to the type of the pointer.

Working of virtual function:

If a class contains virtual keyword the compiler does two things,

1. If object of class is created then VPTR (virtual pointer) is created and inserted as data member of the class to point to VTABLE of the class. For each object created, a new VPTR is inserted as data member of the class.
2. Irrespective of the object created or not, a static array of pointer called VTABLE where each cell contain the address of each virtual function contained in the class.

Note

**Vtable:** A table of function pointers. It is maintained per class.  
**Vptr:** A pointer to vtable. It is maintained per object

**Disadvantage:**

* At runtime the machine has to decide which function call needs to be made, this degrades the performance.
* Each object has its size increased by the amount needed to hold an address.
* For each class, the compiler creates a table of addresses of virtual functions.
* For each function call, there is an extra step of looking up the address on the table.

**Program:**

**#01\_Pgm**

#include<iostream>

class Animal

{

public:

Animal()

{std::cout << "Animal const\n";}

~Animal()

{std::cout << "Animal dest\n";}

virtual void eat()

{std::cout << "Animal eat generic food\n";}

};

class Cat: public Animal

{

public:

Cat()

{std::cout << "Cat Const\n";}

~Cat()

{std::cout << "Cat dest\n";}

void eat()

{std::cout << "Cat eat rat\n";}

};

void func(Animal \*x)

{x->eat();}

int main()

{

Animal \*a = new Animal();

Cat \*c = new Cat();

//a->eat();

//c->eat();

func(a);

func(c);

delete a;

delete c;

}

***Output***

Animal const

Animal const

Cat Const

Animal eat generic food

Cat eat rat

Animal dest

Cat dest

Animal dest

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

**#02\_Pgm**

#include<iostream>

class A{

public:

void f(){

std::cout << "A::f()" << std::endl;

}

};

class B: public A{

public:

void f(){

std::cout << "B::f()\n";

}

};

class C: public B{

public:

void f(){

std::cout << "C::f()\n";

}

};

class D:public C{

public:

// Empty class

};

int main()

{

A\* a = new A();

B\* b = new B();

C\* c = new C();

D\* d = new D();

a->f();

b->f();

c->f();

d->f();

((B\*)c)->f();

((A\*)c)->f();

((A\*)b)->f();

return 0;

}

**Output**

A::f()

B::f()

C::f()

C::f()

B::f()

A::f()

A::f()

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

**#04\_Pgm**

#include<iostream>

class Base{

public:

char\* name;

void display(){ //virtual

std::cout << "name " << name << std::endl;

}

};

class Derived:public Base

{

public:

char\* name;

void display(){

std::cout << "name: " << name << ", " << Base::name << "\n";

}

};

int main()

{

Derived d;

d.name = "Derived Class";

d.Base::name = "Base Class";

Derived\* dptr = &d;

Base\* bptr = &d;

bptr->display();

return 0;

}

**Output**

name Base Class

When used virtual in base class for the display(),

name: Derived Class, Base Class

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

**#05\_Pgm**

#include<iostream>

class A{

public:

virtual void f(){

std::cout << "\nA::f()\n";

}

};

class B: public A{

public:

void f(){

std::cout << "\nB::f()\n";

}

};

class C: public B{

public:

void f(){

std::cout << "\nC::f()\n";

}

};

class D:public C{};

int main()

{

A\* a = new A();

B\* b = new B();

C\* c = new C();

a->f();

b->f();

c->f();

((B\*)c)->f();

((A\*)c)->f();

((A\*)b)->f();

return 0;

}

**Output**

A::f()

B::f()

C::f()

C::f()

C::f()

B::f()

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

**#06\_Pgm**

class Base {

public:

void f();

virtual void vf();

};

class Derived : public Base {

public:

void f();

void vf();

};

#include <iostream>

using namespace std;

void Base::f() {

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

}

void Base::vf() {

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

}

void Derived::f() {

cout << "Derived f()" << endl;

}

void Derived::vf() {

cout << "Derived vf()" << endl;

}

int main()

{

Base b1;

Derived d1;

b1.f();

b1.vf();

d1.f();

d1.vf();

Derived d2; // Derived object

Base\* bp = &d2; // Base pointer to Derived object

bp->f(); // Base f()

*bp->vf();* // which vf()?

return 0;

}

**Output:**

Base f()

Base vf()

Derived f()

Derived vf()

Base f()

Derived vf()

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

**#07\_Pgm**

#include <iostream>

class Base

{

public:

void f() {std::cout << "Base::f()\n";}

// virtual

void vf(){std::cout << "Base::vf()\n";};

};

class Derived: public Base

{

public:

void f() {std::cout <<"Derived::f()\n";}

void vf(){std::cout <<"Derived::vf()\n";};

};

int main()

{

Base b;

Base \*pb = &b;

Derived d;

Derived \*pd = &d;

Base \*pbd = &d;

b.f();

d.f();

pb->f();

pd->f();

pbd->f();

pbd->vf();

return 0;

}

**Output:**

Base::f()

Derived::f()

Base::f()

Derived::f()

Base::f()

Base::vf()

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

**#08\_Pgm**

#include <iostream>

#include <vector>

using namespace std;

class A

{

public:

A(int n = 0) : m(n) {}

public:

virtual int getVal() const {

cout << "A::getVal() = ";

return m;

}

virtual ~A() { }

protected:

int m;

};

class B : public A

{

public:

B(int n = 0) : A(n) {}

public:

int getVal() const {

cout << "B::getVal() = ";

return m + 1;

}

};

int main()

{

const A a(1);

const B b(3);

const A \*pA[2] = { &a, &b };

cout << pA[0]->getVal() << endl;

cout << pA[1]->getVal() << endl;

vector<A> vA;

vA.push\_back(a);

vA.push\_back(b);

vector<A>::const\_iterator it = vA.begin();

cout << it->getVal() << endl;

cout << (it + 1)->getVal() << endl;

return 0;

}

**Output:**

A::getVal() = 1

B::getVal() = 4

A::getVal() = 1

A::getVal() = 3

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

**Virtual Destructor:**

It should be defined to ensure proper destructor is called for the classes derived from it. If a base class contains virtual function then its a must ot use virtual destructor instead of destructor.

**Why virtual destructor:**

If we call delete on a base pointer which points to a derived class object, the base class destructor get called first for non-virtual function.

To ensure that the proper destructor is called if this class is derived from an object of the derived class and is deallocated using object expression in which the static type refers to the base class.

**When to use virtual destructor:**

1. If a class has a virtual function, It’s likely to be used as a base class.

2. If its a base class, its derived class is likely to be allocated using the new.

3. If a derived class is allocated using new and manipulated through a pointer to its base, it is likely to be deleted via pointer to base.

**Program:**

**#09\_Pgm (Virtual Destructor)**

#include <iostream>

using namespace std;

class Base

{

public:

Base() {

cout << "Base Constructor \n" ;

}

~Base() {

cout << "Base Destructor \n" ;

}

};

class Derived : public Base

{

public:

Derived(string s):str(s) {

cout << "Derived Constructor \n" ;

}

~Derived() {

cout << "Derived Destructor \n" ;

}

private:

string str;

};

int main()

{

Base \*pB = new Derived("derived");

delete pB;

}

**Output**

Base Constructor

Derived Constructor

Base Destructor

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

**#10\_Pgm**

#include <iostream>

using namespace std;

class Base{

protected:

int myInt;

public:

Base(int n):myInt(n){

cout << "Base Ctor\n";

}

virtual void print() const = 0;

virtual ~Base(){

cout << "Base Dtor" << endl;

}

};

class Derived: public Base {

public:

Derived(int n = 0):Base(n) {

str = new char[100];

myInt = n;

cout << "Derived Ctor myInt" << endl;

}

void print()const{

cout << "Derived print(): myInt = "<< myInt << endl;

}

~Derived(){

cout << "Derived Dtor" << endl;

delete [] str;

}

private:

char \*str;

};

int main()

{

Base \*pB = new Derived(2010);

pB->print();

delete pB;

return 0;

}

**Output:**

Base Ctor

Derived Ctor myInt

Derived print(): myInt = 2010

Derived Dtor

Base Dtor

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

**Virtual destructor:**

*Constructors can't be virtual!*

"Creating a derived object invokes a derived class constructor, not a base class constructor. The derived class constructor then uses a base class constructor, but the sequence is distinct from the inheritance mechanism. Therefore, a derived class doesn't inherit the base class constructors, so usually there's not much point to making them virtual, anyway."

**Pure virtual function:**

It is essential for creating abstract classes, It can be used to declare functions with no implementations.

Example:

virtual void fun() = 0; // A pure virtual function

Here the suppose the fun is present in the base class, then the base class becomes abstract class. Hence the instance cannot be created for the base class and only the derived class can create instance if the pure virtual function has a been defined in the derived class.

1. They cannot instantiate an object of this class - they should create a child class from it.

2. They must override all pure virtual functions in the child class, or they will not be able to instantiate the child class.

**Pure virtual destructor:**

A pure virtual destructor is legal in c++ provided a functional body for it.

class Interface {

public:

//pure virtual destructor declaration

virtual ~Interface() = 0;

}

//definition of a pure virtual destructor; should always be empty

Interface::~Interface() {}

Constructor and destructors are special member function which cannot be present without the definition for the function.

A pure virutal destructor causes a class to be abstract and object cannot be created.

**What is a abstract class:**

It a template class through the derived class the implementations are performed.