### Virtual Functions

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

#### Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

#### Problem without Virtual Keyword

class Base

{

public:

void show() // belongs to base class//

{

cout << "Base class";

}

};

class Derived:public Base

{

public:

void show() //belongs to derived class//

{

cout << "Derived Class";

}

}

int main()

{

Base\* b; *//Base class pointer*

Derived d; *//Derived class object*

b = &d;

b->show(); *//Early Binding Ocuurs*

}

Output : Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

#### Using Virtual Keyword

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

class Base

{

public:

**virtual** void show()

{

cout << "Base class";

}

};

class Derived:public Base

{

public:

void show()

{

**cout << "Derived Class";**

}

}

int main()

{

Base\* b; *//Base class pointer*

Derived d; *//Derived class object*

b = &d;

b->show(); *//Late Binding Occurs*

}

Output : Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

#### Using Virtual Keyword and Accessing Private Method of Derived class

We can call **private** function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

#include

using namespace std;

class A

{

public:

**virtual** void show()

{

cout << "Base class\n";

}

};

class B: public A

{

private:

**virtual** void show()

{

cout << "Derived class\n";

}

};

int main()

{

A \*a;

B b;

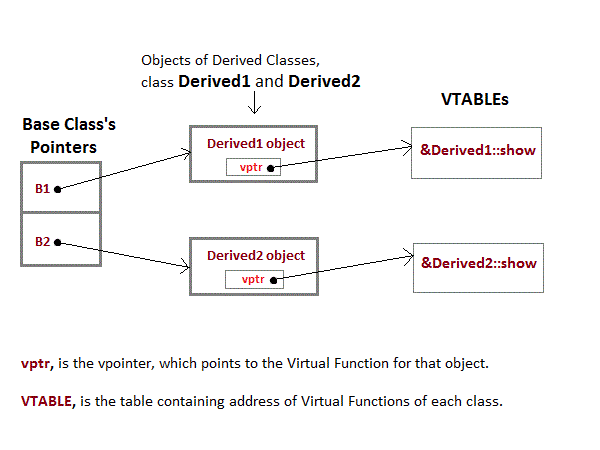
a = &b;

a **->** show();

}

Output : Derived class

#### Mechanism of Late Binding



To accomplich late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

#### Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function.

**Overriding**

1)Methods name and signatures must be same.

2)Overriding is the concept of runtime polymorphism

3)When a function of base class is re-defined in the derived class called as Overriding

4)It needs inheritance.

5)Method should have same data type.

6)Method should be public.

**Overloading**

1)Having same method name with different Signatures.

2)Overloading is the concept of compile time polymorphism

3)Two functions having same name and return type, but with different type and/or number of arguments is called as Overloading

4)It doesn't need inheritance.

5)Method can have different data types

6)Method can be different access specifies

Example programs:

#include <iostream>

using namespace std;

class base {

public:

virtual void vfunc() {

cout << "This is base's vfunc().\n";

}

};

class derived1 : public base {

public:

void vfunc() {

cout << "This is derived1's vfunc().\n";

}

};

class derived2 : public base {

public:

void vfunc() {

cout << "This is derived2's vfunc().\n";

}

};

int main()

{

base \*p, b;

derived1 d1;

derived2 d2;

// point to base

p = &b;

p->vfunc(); // access base's vfunc()

// point to derived1

p = &d1;

p->vfunc(); // access derived1's vfunc()

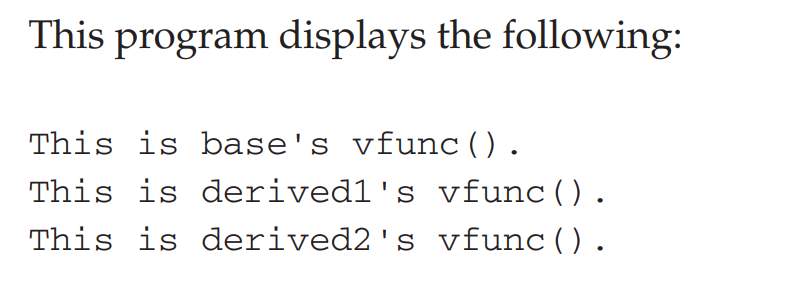
// point to derived2

p = &d2;

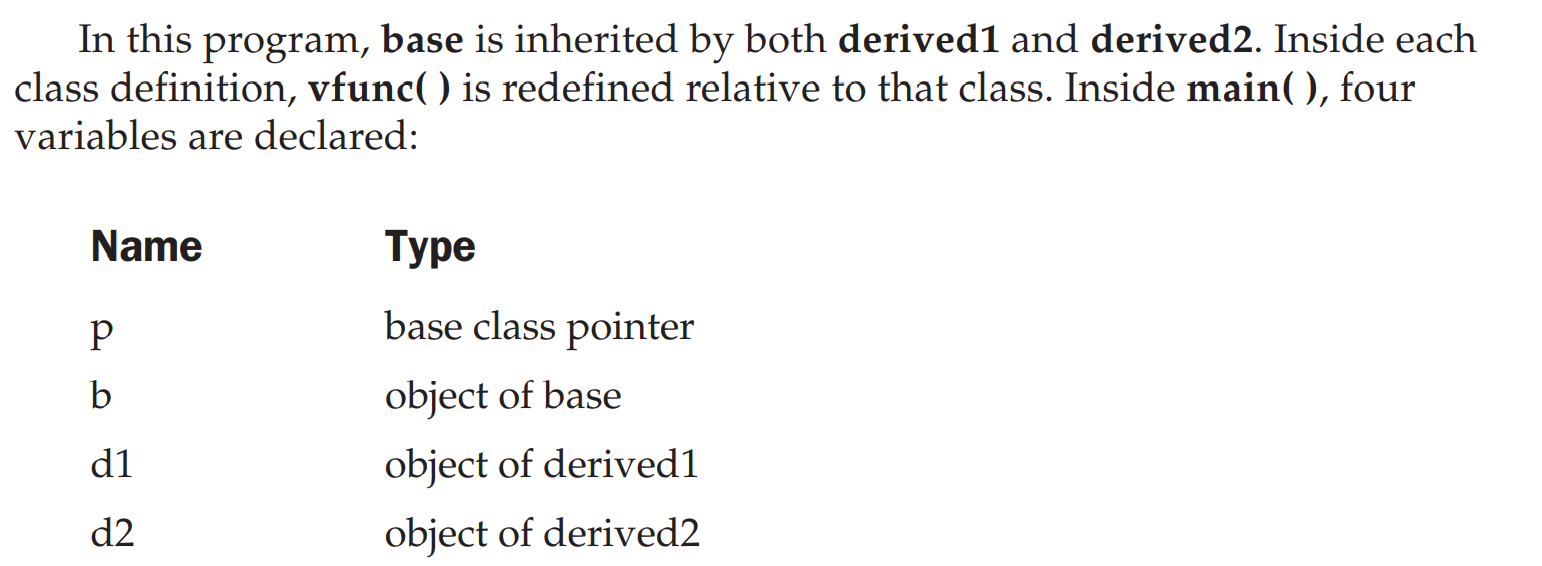
p->vfunc(); // access derived2's vfunc()

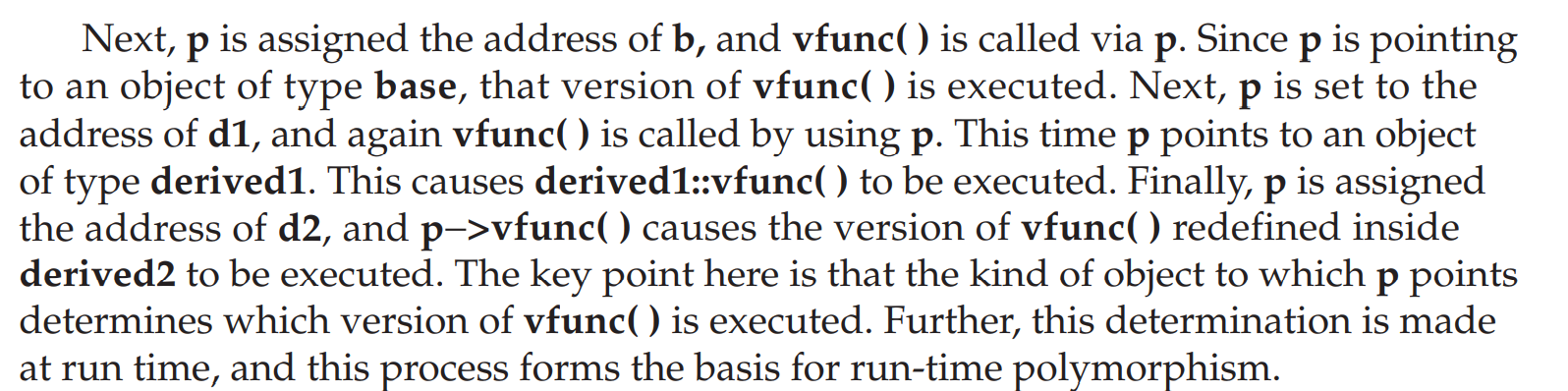
return 0;

}



As the program illustrates, inside base, the virtual function vfunc( ) is declared. Notice that the keyword virtual precedes the rest of the function declaration. When vfunc( ) is redefined by derived1 and derived2, the keyword virtual is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)





Calling a Virtual Function Through a Base Class Reference

#include <iostream>

using namespace std;

class base {

public:

virtual void vfunc() {

cout << "This is base's vfunc().\n";

}

};

class derived1 : public base {

public:

void vfunc() {

cout << "This is derived1's vfunc().\n";

}

};

class derived2 : public base {

public:

void vfunc() {

cout << "This is derived2's vfunc().\n";

}

};

int main()

{

base \*p, b;

derived1 d1;

derived2 d2;

// point to base

p = &b;

p->vfunc(); // access base's vfunc()

// point to derived1

p = &d1;

p->vfunc(); // access derived1's vfunc()

// point to derived2

p = &d2;

p->vfunc(); // access derived2's vfunc()

return 0;

}

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

#include <iostream>

using namespace std;

class base {

public:

virtual void vfunc() {

cout << "This is base's vfunc().\n";

}

};

class derived1 : public base {

public:

void vfunc() {

cout << "This is derived1's vfunc().\n";

}

};

/\* derived2 inherits virtual function vfunc()

from derived1. \*/

class derived2 : public derived1 {

public:

// vfunc() is still virtual

void vfunc() {

cout << "This is derived2's vfunc().\n";

}

};

int main()

{

base \*p, b;

derived1 d1;

derived2 d2;

// point to base

p = &b;

p->vfunc(); // access base's vfunc()

// point to derived1

p = &d1;

p->vfunc(); // access derived1's vfunc()

// point to derived2

p = &d2;

p->vfunc(); // access derived2's vfunc()

return 0;

}

Virtual Functions Are Hierarchical

when a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which derived2 does not override vfunc( ):

Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast. The opposite of early binding is late binding.

As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times

## 

## #include <iostream>

## using namespace std;

## class number {

## protected:

## int val;

## public:

## void setval(int i) { val = i; }

## // show() is a pure virtual function

## virtual void show() = 0;

## };

## class hextype : public number {

## public:

## void show() {

## cout << hex << val << "\n";

## }

## };

## class dectype : public number {

## public:

## void show() {

## cout << val << "\n";

## }

## };

## class octtype : public number {

## public:

## void show() {

## cout << oct << val << "\n";

## }

## };

## int main()

## {

## dectype d;

## hextype h;

## octtype o;

## d.setval(20);

## d.show(); // displays 20 - decimal

## h.setval(20);

## h.show(); // displays - hexadecimal

## o.setval(20);

## o.show(); // displays - octal

## return 0;

## }