virtual function table analysis

C++ virtual function table analysis

Chen Hao
http://blog.csdn.net/haoel

ord

f virtual functions in C++ is mainly to realize the mechanism of polymorphism. Regarding hism, in a nutshell, it uses the pointer of the parent type to point to the instance of its , and then calls the member function of the actual subclass through the pointer of the lass. This technique allows the pointer of the parent class to have "multiple shapes", a generic technique. The so-called generic technology, to put it bluntly, is trying to use code to implement a variable algorithm. For example: template technology, RTTI technology, function technology, either try to do resolution at compile time, or try to do runtime on.

g the use of virtual functions, I will not elaborate too much here. You can look at the C++ books. In this article, I just want to give you a clear analysis of the tation mechanism of virtual functions.

e, the same articles have appeared on the Internet, but I always feel that these articles very easy to read, with large sections of code, no pictures, no detailed descriptions, no ons, and no inferences. It's not good for learning and reading, so that's why I want to is article. I also hope that you will give me more opinions.

o home, let us enter the world of virtual functions together.

al function table

ho knows C++ should know that virtual functions (Virtual Function) are implemented through a function table (Virtual Table). Referred to as V-Table . In this table, we mainly need the table of the virtual function of a class. This table solves the problem of inheritance and , and ensures that its content truly reflects the actual function. In this way, in an of a class with virtual functions, this table is allocated in the memory of this instance, we use the pointer of the parent class to operate a subclass, this virtual function table important. Now, it is like a map, indicating the actual function that should be called.

focus on this virtual function table. The C++ compiler should ensure that the pointer to ual function table exists in the frontmost position of the object instance (this is to he highest performance of fetching the virtual function table - if there are multiple f inheritance or multiple inheritance)). This means that rough the address of the object instance, and then we can traverse the function pointer and corresponding function.

all this talk, I can tell you're probably more dizzy now than you were before. It doesn't the following is an actual example, I believe you will understand it at a glance.

we have a class like this:

```
ss Base {
 public :
        virtual void f() { cout << "Base::f" << endl; }
        virtual void g() { cout << "Base::g" << endl; }
        virtual void h() { cout << "Base::h" << endl; }
```

g to the above statement, we can get the virtual function table through the instance of 'e is the actual routine:

```
›edef void (*Fun)( void );

e b;

 pFun = NULL;

t << " Address of virtual function table: " << ( int *)(&b) << endl;
t << " Virtual function table - address of the first function: " << ( int *)*( int *)(&b) << endl;

ıvoke the first virtual function
n = (Fun)*(( int *)*( int *)(&b));
n();
```

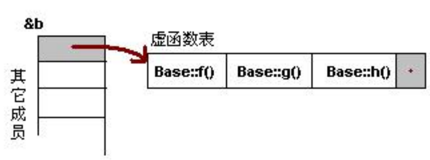
al running results are as follows: (Windows XP+VS2003, Linux 2.6.22 + GCC 4.1.3)

```
rtual function table address: 0012FED4
rtual function table - first function address: 0044F148
se::f
```

this example, we can see that we can get the address of the virtual function table by converting &b to int *, and then, by taking the address again, we can get the address of t virtual function, which is Base::f (), which is verified in the program above (casting the function pointer). Through this example, we can know that if we want to call Base::g() and , the code is as follows:

```
ın)*(( int *)*( int *)(&b)+0); // Base::f()
ı)*(( int *)*( int *)(&b)+1); // Base::g()
ı)*(( int *)*( int *)(&b)+2); // Base::h()
```

ld understand by now. What? Still a little dizzy. Also, such code looks too messy. No let me draw a picture to explain. As follows:



the above figure, I added an extra node at the end of the virtual function table, which is node of the virtual function table, just like the end cha

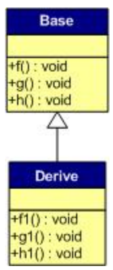
e virtual function table end of. The value of this end fl

s. Under WinXP+VS2003, this value is NULL. Under Ubuntu 7.10 + Linux 2.6.22 + GCC 4.1.3, if e is 1, it means there is the next virtual function table, and if the value is 0, it means virtual function table.

'll explain what the virtual function table looks like with "no coverage" and "with ", respectively. Virtual functions without overriding the parent class are meaningless. The son why I'm going to talk about the case without coverage is to give a comparison. In on, we can more clearly know the specific implementation of its internal.

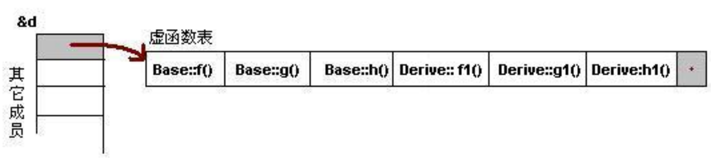
al inheritance (no virtual function override)

t's take a look at what the virtual function table looks like during inheritance. Suppose an inheritance relationship as follows:



t in this inheritance relationship, the subclass does not overload any functions of the ss. Then, in an instance of a derived class, its virtual function table looks like this:

ple: Derive d; the virtual function table is as follows:

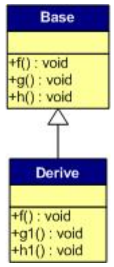


ee the following points:
rtual functions are placed in the table in the order in which they are declared.
e virtual function of the parent class is in front of the virtual function of the child ass.

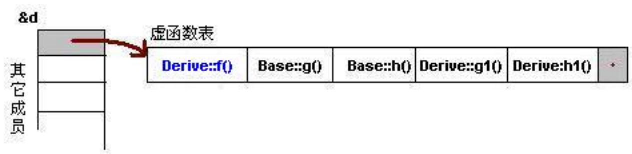
e that you are smart enough to refer to the previous program to write a program to verify.

al inheritance (with virtual function override)

vious to override the virtual function of the parent class, otherwise, the virtual function meaningless. Next, let's take a look, if there is a virtual function in the subclass that s the virtual function of the superclass, what will it look like? Suppose, we have an nce relationship like the following.



to let everyone see the effect of being inherited, in the design of this class, I only one function of the parent class: f(). Then, for an instance of a derived class, its function table will look like the following:



ee the following points from the table,
e overwrittenf()function is placed in the position of the original parent class virtual nction in the virtual table.
ıctions that are not overridden remain.

can see that for a program like the following,

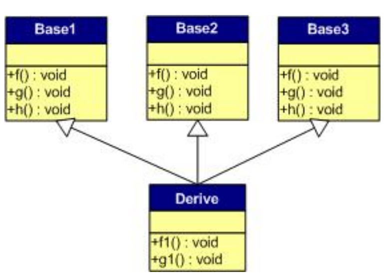
```
e *b = new Derive();

f();
```

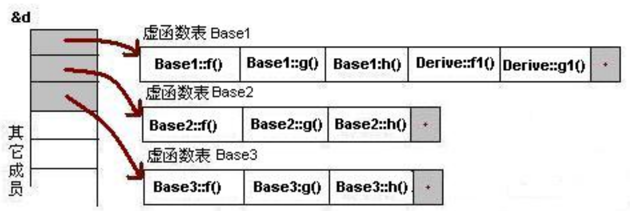
tion of f() in the virtual function table pointed to by b has been replaced by the address of the) function, so when the actual call occurs, Derive::f() is called. This achieves polymorphism.

ɔle inheritance (no virtual function override)

t's take a look at the situation in multiple inheritance, assuming that there is an nce relationship of the following class. Note: The subclass does not override the function uperclass.



virtual function table in the subclass instance, it looks like this:



ee that:

h parent class has its own virtual table.

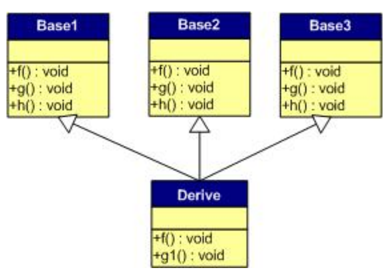
member functions of the subclass are placed in the table of the first parent class. (The
-called first parent class is judged according to the declaration order)

to solve the problem that pointers of different parent class types point to the same
instance, and can call the actual function.

ple inheritance (with virtual function override)

ke a look again, if virtual function coverage occurs.

ge below, we have overridden the f() function of the parent class in the child class .



owing is a diagram for the virtual function table in the subclass instance:

e that the position of f() in the virtual function table of the three parent classes is replaced by
tion pointer of the child class. In this way, we can point to the subclass from any
ly typed superclass and call f() of the subclass. like:

```
ive d;
e1 *b1 = &d;
e2 *b2 = &d;
e3 *b3 = &d;
>f(); //Derive::f()
>f(); //Derive::f()
>f(); //Derive::f()

>g(); //Base1::g()
>g(); //Base2::g()
```

```
>g(); //Base3::g()
```

y

me I write an article on C++, it is inevitable to criticize C++. This article is no n. Through the above description, I believe we have a more detailed understanding of the function table. Water can carry a boat or capsize it. Next, let's see what bad things we ith virtual function tables.

access the virtual function of the subclass itself through the pointer of the pe

that it is a meaningless thing that the subclass does not overload the virtual function of rclass. Because polymorphism is also based on function overloading. Although in the above e can see that there is a Derive virtual function in the virtual table of Base1, it is le for us to use the following statement to call the subclass's own virtual function:

```
se1 *b1 = new Derive();
>f1();   // Compile error
```

mpt to use the parent class pointer to call the member function of the subclass that t override the parent class will be regarded as illegal by the compiler, so such a cannot be compiled at all. But at runtime, we can access the virtual function table through to achieve behavior that violates C++ semantics. (For an attempt at this, by reading the the appendix below, I believe you can do it)

ss **non-public** virtual functions

ion, if the virtual function of the parent class is private or protected, but these non- public functions will also exist in the virtual function table, so we can also use the method of g the virtual function table to access these non-public virtual functions virtual function, easy to do.

```
ss Base {
private :
    virtual void f() { cout << "Base::f" << endl; }

ss Derive : public Base{

edef void (*Fun)( void );

d main() {
Derive d;
Fun  pFun = (Fun)*(( int *)*( int *)(&d)+ 0 );
pFun();
```

ıding remarks

magic language. For programmers, we never seem to know w

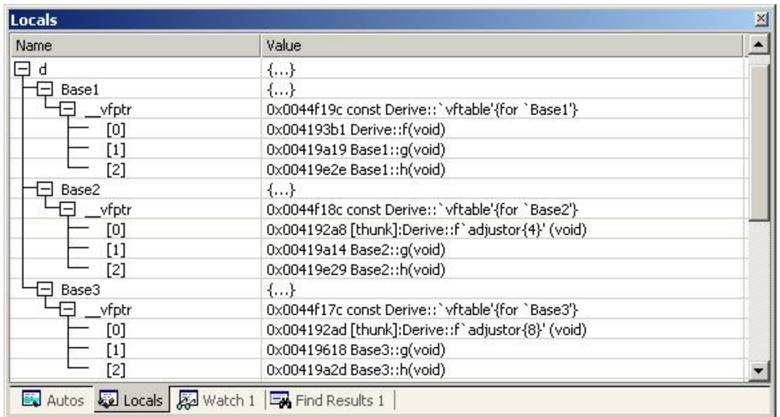
s. To be familiar with this language, we must understand

d those dangerous things in C++. Otherwise, it's a programming language that shoots itself in
.

he end of the article, let me introduce myself. I have been engaged in software research lopment for ten years. I am currently the technical director of software development. In technology, I mainly focus on Unix / C / C++. related stuff. In terms of management, he is team building, technology trend analysis, and project management. Welcome everyone to ate with me, my MSN and Email are: haoel@hotmail.com

dix 1: View the virtual function table in VC

xpand the instance of the class in the Debug state of the VC IDE environment to see the function table (not very complete)



dix II: Routines

an example of a virtual function table access for multiple inheritance:

```
ostream>
space std;

{

ual void f() { cout << "Base1::f" << endl; }
ual void g() { cout << "Base1::g" << endl; }
ual void h() { cout << "Base1::h" << endl; }



{

ual void f() { cout << "Base2::f" << endl; }
ual void g() { cout << "Base2::g" << endl; }
ual void h() { cout << "Base2::h" << endl; }


{

ual void f() { cout << "Base3::f" << endl; }
ual void g() { cout << "Base3::g" << endl; }
ual void h() { cout << "Base3::h" << endl; }



 : public Base1, public Base2, public Base3 {

ual void f() { cout << "Derive::f" << endl; }
```

```
ual void g1() { cout << "Derive::g1" << endl; }

 (*Fun)( void );

 pFun = NULL;

ive d;
** pVtab = ( int **)&d;

Base1's vtable
Fun = (Fun)*((int*)*(int*)((int*)&d+0)+0);
n = (Fun)pVtab[ 0 ][ 0 ];
n();

Fun = (Fun)*((int*)*(int*)((int*)&d+0)+1);
n = (Fun)pVtab[ 0 ][ 1 ];
n();

Fun = (Fun)*((int*)*(int*)((int*)&d+0)+2);
n = (Fun)pVtab[ 0 ][ 2 ];
n();

erive's vtable
Fun = (Fun)*((int*)*(int*)((int*)&d+0)+3);
n = (Fun)pVtab[ 0 ][ 3 ];
n();

he tail of the vtable
n = (Fun)pVtab[ 0 ][ 4 ];
t<<pFun<<endl;

Base2's vtable
Fun = (Fun)*((int*)*(int*)((int*)&d+1)+0);
n = (Fun)pVtab[ 1 ][ 0 ];
n();

Fun = (Fun)*((int*)*(int*)((int*)&d+1)+1);
n = (Fun)pVtab[ 1 ][ 1 ];
n();

n = (Fun)pVtab[ 1 ][ 2 ];
n();

he tail of the vtable
n = (Fun)pVtab[ 1 ][ 3 ];
t<<pFun<<endl;

Base3's vtable
Fun = (Fun)*((int*)*(int*)((int*)&d+1)+0);
n = (Fun)pVtab[ 2 ][ 0 ];
n();

Fun = (Fun)*((int*)*(int*)((int*)&d+1)+1);
n = (Fun)pVtab[ 2 ][ 1 ];
```

```
n();

n = (Fun)pVtab[ 2 ][ 2 ];
n();

ne tail of the vtable
n = (Fun)pVtab[ 2 ][ 3 ];
t<<pFun<<endl;

urn 0 ;
```

indicate the author and source when reprinting. Please do not use for commercial s without permission)

articles, please visit my Blog: http://blog.csdn.net/haoel

al function latest release weixin_45138932's blog 1667
function declared as virtual in a base class and redefined in one or more derived classes is called a virtual function ...

ıction virtual function table qq_41078889's blog 2355
ction is a specific form of object-oriented programming function , and it is an effective mechanism for C++ to realize ...

explanation of C++ virtual functions qq_42048450's blog 10,000+
ırtual functions ? 1.1 Definition of virtual functions Virtual functions are used when implementing C++ polymorphism...

explanation of C++ virtual functions smartDMer's column 4631
ıe role of virtual functions in C++ is mainly to realize the mechanism of polymorphism. Regarding polymorphism, in ...

ept and use of virtual functions Technical Notes 2133
uces virtual functions in order to use polymorphism . The role of virtual functions is to allow redefinition of functions ...

and use of virtual functions 20164225's Blog 3471
from: http://c.biancheng.net/cpp/biancheng/view/244.html We know that it is impossible to define two functions with ...

al function table Linux Ape 3042
mainly analyzes and summarizes the virtual function table . Virtual functions have been summarized in previous ar...

popular articles

Debugging programs with GDB (1) 647990

Write Makefile with me (1) 613780

C++ virtual function table parsing 481701

Actually Unix is very simple 160759

Memory layout of C++ objects (on)

Table of contents