虚函数和指针结合使用可以产生最大的效果。

1. 非虚函数是静态绑定的；

2. 虚函数可能(may)是动态绑定的；

3. 一个指针实际上可能指向了衍生类对象，当虚函数被指针调用时，绑定哪个函数取决于被指对象的类，而不是指针的类型；

4. 所以，虚函数根据执行函数的类型（接收者，the receiver）来绑定。

静态绑定和动态绑定的区别：

1. 动态绑定：只有在程序运行时才能决定调用哪个函数；

2. 静态绑定：编译时就可以决定调用哪个函数；

3. C++中，如果接收者是个指针则虚函数动态绑定（例如：如果 f(...)为虚函数，则 pointer->f(...)动态绑定）；

4. 其他函数都是静态绑定。

虚函数和重载函数的区别：（转自：http://blog.csdn.NET/livelylittlefish/article/details/2171515，我觉得这个程序解释的很清楚）

1. 重载函数在类型和参数数量上一定不相同，而重定义的虚函数则要求参数的类型和个数、函数返回类型相同；

2. 虚函数必须是类的成员函数，重载的函数则不一定是这样；

3. 构造函数可以重载，但不能是虚函数，析构函数可以是虚函数。

例 1：

```
#include <iostream.h>
class CBase
{
public:
    virtual int func(unsigned char ch) {return --ch;}
};
class CDerive : public CBase
{
    int func(char ch) {return ++ch;}   //此为函数重载
};
void main()
{   CBase *p=new CDerive;
    int n=p->func(40);       //调用基类的 func()
```

```
    cout<<" the result is : "<<n<<endl;
}
```

运行结果：

the result is : 39

例 2：

```
#include <iostream.h>
class CBase
{
public:
    virtual int func(unsigned char ch) {return --ch;}
};
class CDerive : public CBase
{
    int func(unsigned char ch) {return ++ch;}      //此为虚函数
};
void main()
{   CBase *p=new CDerive;
    int n=p->func(40);        //调用派生类的 func()
    cout<<" the result is : "<<n<<endl;
}
```

运行结果：

the result is : 41

这个例子可以验证

“ 3. 一个指针实际上可能指向了衍生类对象，当虚函数被指针调用时，绑定哪个函数取决于被指对象的类，而不是指针的类型；”

虚函数和指针结合使用可以产生最大的效果。

1. 非虚函数是静态绑定的；

2. 虚函数可能(may)是动态绑定的；

3. 一个指针实际上可能指向了衍生类对象，当虚函数被指针调用时，绑定哪个函数取决于被指对象的类，而不是指针的类型；

4. 所以，虚函数根据执行函数的类型（接收者，the receiver）来绑定。

静态绑定和动态绑定的区别：

1. 动态绑定：只有在程序运行时才能决定调用哪个函数；

2. 静态绑定：编译时就可以决定调用哪个函数；

3. C++中，如果接收者是个指针则虚函数动态绑定（例如：如果 f(...)为虚函数，则 pointer->f(...)动态绑定）；

4. 其他函数都是静态绑定。

虚函数和重载函数的区别：（转自：http://blog.csdn.NET/livelylittlefish/article/details/2171515，我觉得这个程序解释的很清楚）

1. 重载函数在类型和参数数量上一定不相同，而重定义的虚函数则要求参数的类型和个数、函数返回类型相同；

2. 虚函数必须是类的成员函数，重载的函数则不一定是这样；

3. 构造函数可以重载，但不能是虚函数，析构函数可以是虚函数。

例 1：

```
#include <iostream.h>
class CBase
{
public:
    virtual int func(unsigned char ch) {return --ch;}
};
class CDerive : public CBase
{
    int func(char ch) {return ++ch;}   //此为函数重载
};
void main()
{   CBase *p=new CDerive;
    int n=p->func(40);       //调用基类的 func()
```

```
    cout<<" the result is : "<<n<<endl;
}
```

运行结果：

the result is : 39

例 2：

```
#include <iostream.h>
class CBase
{
public:
    virtual int func(unsigned char ch) {return --ch;}
};
class CDerive : public CBase
{
    int func(unsigned char ch) {return ++ch;}      //此为虚函数
};
void main()
{   CBase *p=new CDerive;
    int n=p->func(40);        //调用派生类的 func()
    cout<<" the result is : "<<n<<endl;
}
```

运行结果：

the result is : 41

这个例子可以验证

“ 3. 一个指针实际上可能指向了衍生类对象，当虚函数被指针调用时，绑定哪个函数取决于被指对象的类，而不是指针的类型；”

1、多态的实现机制

C++在基类中声明一个带关键之 Virtual 的函数，这个函数叫虚函数；它可以在该基类的派生类中被重新定义并被赋予另外一种处理功能。通过指向指向派生类的基类指针或引用调用虚函数，编译器可以根据指向对象的类型在运行时决定调用的目标函数。这就实现了多态。

2、实例

[cpp] view plain copy

```cpp
#include<iostream>
using namespace std;

class Base
{
public:
virtual void fun1 () {cout<<" printf base fun1!" <<endl;}
virtual void fun2 () {cout<<" printf base fun2!" <<endl;}
private:
int m_data1;
} ;

class Derive: public Base
{
public:
void fun1 () {cout<<" printf derive fun1!" <<endl;}
void fun3 () {cout<<" printf derive fun3" <<endl;}
private:
int m_data2;
} ;

int main ()
{
Base *pBase=new Derive;
Derive a;
pBase->fun1 () ;
pBase->fun2 () ;
a.fun3 () ;
return 0;
```

```
30. }
```

3、底层机制

在每一个含有虚函数的类对象中，都含有一个 VPTR，指向虚函数表。

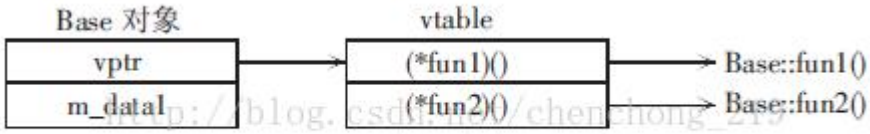


图 2 Base 对象内存空间示意图

派生类也会继承基类的虚函数，如果宅派生类中改写虚函数，虚函数表就会受到影响；表中元素所指向的地址不是基类的地址，而是派生类的函数地址。

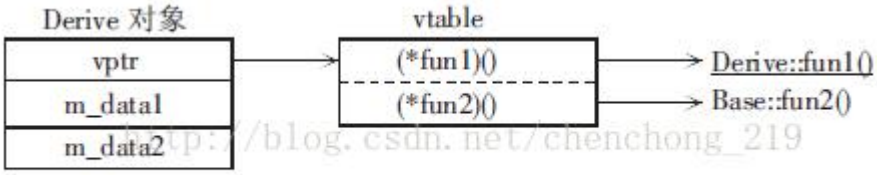


图 3 Derive 对象内存空间示意图

当执行语句 pBase->fun1()时，由于 PBase 指向的是派生类对象，于是就调用的 Deriver::fun1()。

4、多重继承

[cpp] view plain copy

```
1. #include<iostream_h>
2. class base1
3. {
4. public:
5. virtual void vn(){}
6. private:
7. inti;
8. );
9. class base2
10. {
11. public:
12. virtual void vf2(){}
```

```
13. private:
14. intj;
15. );
16. class derived: public base 1, public base2
17. {
18. public:
19. virtual void vf3(){}
20. private:
21. int k:
22. );
23. void main()
24. {
25. derivedd:
26. base1 p1;
27. base2 p2;
28. p1=&d; p2 &d:
29. p1->vf1();
30. p2->vf2();
31. }
```

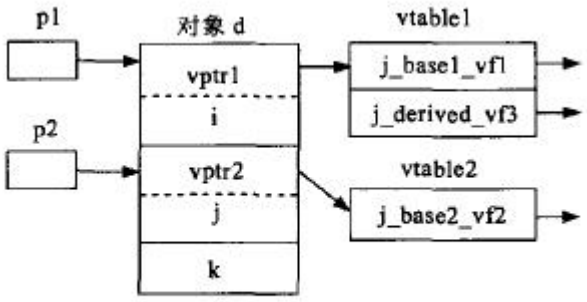


图 3　多继承下虚函数的实现机制

如果一个类具有多个包含虚函数的父类，编译器会为它创建多个 VIrtual table，每个 virtual table 中各个虚函数的顺序与相应的父类一样。

C++编译期间的虚函数调用机制

```
class  CMyWnd : public CWindowImpl<CMyWnd>

{

    ...
```

```
};
```

这样作是合法的，因为 C++的语法解释说即使 CMyWnd 类只是被部分定义，类名 CMyWnd 已经被列入递归继承列表，是可以使用的。将类名作为模板类的参数是因为 ATL 要做另一件诡秘的事情，那就是编译期间的虚函数调用机制(传输中的静多态)。

如果你想要了解它是如何工作地，请看下面的例子：

```
template<class T>

class B1

{

public:

   void SayHi()

   {

      T* pT = static_cast<T*>(this);   // 展开模板你就会明白



      pT->PrintClassName();

   }

protected:

   void PrintClassName() { cout << "This is B1"; }

};



class D1 : public B1<D1>

{

   // No overridden functions at all
```

```
};

class D2 : public B1<D2>
{
protected:
    void PrintClassName() { cout << "This is D2"; }
};

main()
{
    D1 d1;
    D2 d2;

    d1.SayHi();     // prints "This is B1"
    d2.SayHi();     // prints "This is D2"
}
```

这句代码 static_cast<T*>(this) 就是窍门所在。它根据函数调用时的特殊处理将指向 B1 类型的指针 this 指派为 D1 或 D2 类型的指针，因为模板代码是在编译其间生成的，所以只要编译器生成正确的继承列表，这样指派就是安全的。（如果你写成：

```
class D3 : public B1<D2>
```

就会有麻烦) 之所以安全是因为 this 对象只可能是指向 D1 或 D2（在某些情况下）类型的对象，不会是其他的东西。注意这很像 C++的多态性（polymorphism），只是 SayHi()方法不是虚函数。

要解释这是如何工作的，首先看对每个 SayHi()函数的调用，在第一个函数调用，对象 B1 被指派为 D1，所以代码被解释成：

```
void B1<D1>::SayHi()

{

   D1* pT = static_cast<D1*>(this);   // 为什么可以转呢, 因为 D1 继承 B1



   pT->PrintClassName();

}
```

由于 D1 没有重载 PrintClassName()，所以查看基类 B1，B1 有 PrintClassName()，所以 B1 的 PrintClassName()被调用。

现在看第二个函数调用 SayHi()，这一次对象被指派为 D2 类型，SayHi()被解释成：

```
void B1<D2>::SayHi()

{

   D2* pT = static_cast<D2*>(this);



   pT->PrintClassName();

}
```

这一次，D2 含有 PrintClassName()方法，所以 D2 的 PrintClassName()方法被调用。

这种技术的有利之处在于：

- 不需要使用指向对象的指针。
- 节省内存，因为不需要虚函数表。
- 因为没有虚函数表所以不会发生在运行时调用空指针指向的虚函数。
- 所有的函数调用在编译时确定(译者加:区别于 C++的虚函数机制使用的动态编连)，有利于编译程序对代码的优化。

节省虚函数表在这个例子中看起来无足轻重（每个虚函数只有 4 个字节），但是设想一下如果有 15 个基类，每个类含有 20 个方法，加起来就相当可观了。

引出：写个类 A，声明类 A 指针指向 NULL，调用类 A 的方法会有什么后果，编译通过吗，运行会通过吗？

(在 VS2008 与 VC++的情况下) 有错误欢迎批评指正！

```
#include<stdio.h>
#include<iostream>using namespace std;
class base{
    int a;public:
    void fun(){
        printf("base fun\n");
    }
};

int main(){
    base *b=NULL;
    b->fun();
}
```

看到这个的时候，一定以为运行会报错吧。

但是奇迹般的，编译器输出了：base fun

```
#include<stdio.h>
#include<iostream>using namespace std;
class base{
    int a;public:
    virtual void fun(){
        printf("base fun\n");
    }
};

int main(){
    base *b=NULL;
    b->fun();
}
```

在看这个代码，还以为会输出 base fun 么，又错了，运行报错！

为什么会是这个结果？

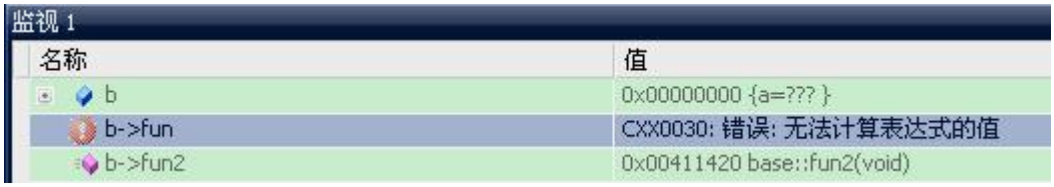
```
#include<stdio.h>
#include<iostream>using namespace std;
class base{
    int a;public:
    virtual void fun(){
        printf("base fun\n");
    }
        void fun2(){
        printf("base fun\n");
    }
};

int main(){
    base *b=NULL;
    b->fun();
    b->fun2();
```

}

可以发现，一个是虚函数，一个普通函数

在观察下内存中得情况：



发现果然虚函数还没在内存中，而 fun2 已经在内存中了

在看看汇编：

```
    base *b=NULL;
004113CE  mov         dword ptr [b],0
    b->fun();
004113D5  mov         eax,dword ptr [b]
004113D8  mov         edx,dword ptr [eax]
004113DA  mov         esi,esp
004113DC  mov         ecx,dword ptr [b]
004113DF  mov         eax,dword ptr [edx]
004113E1  call        eax
004113E3  cmp         esi,esp
004113E5  call        @ILT+315(__RTC_CheckEsp) (411140h)
    b->fun2();
004113EA  mov         ecx,dword ptr [b]
004113ED  call        base::fun2 (411014h)
```

明显发现虚函数的调用比普通函数多了好几个步骤，

ecx 中放的 this 指针，所以 this=0(NULL),但是普通函数 fun2 放在全局内存区，所以可以访问

而虚函数是根据虚函数表寻找的，这时没有虚函数表，自然就没法查到虚函数的地址了

感谢 hoodlum1980 更详细的说明：因为非虚函数的地址对编译期来说“静态”的，也就是函数地址在编译期就已经确定了，实例地址对于非虚函数只是那个 this 指针参数。所以只要不访问类的实例数据就没什么问题。而虚函数的地址，是先到实例的地址前面去查找它的虚函数表所在的地址。然后从虚函数表里取出该函数所对应的元素（虚函数表是一个函数指针数组）来 call 的。（当然一个已知的类的虚函数表的内容也是编译期静态的，但不同类的虚

函数表内容不同，即运行时多态的基础）所以实例如果为 NULL，是个有特殊意义的值，是会触发运行时错误的。

总结：类中的虚函数是动态生成的，由虚函数表的指向进行访问，不为类的对象分配内存，就没有虚函数表就无法访问。

类中的普通函数静态生成，不为类的对象分配内存也可访问。