C++中的虚函数是通过虚函数表来实现，简称V-Table。虚函数表是一个类的虚函数的地址表，存储的是类的虚函数的函数指针。

参考：

http://blog.csdn.net/haoel/article/details/1948051/

虚函数表存放在实例对象的内存中，而且，保存虚函数表的指针通常存放在对象实例中最前面的位置，我们可以通过实例对象来获取这张虚函数表，并调用相应的函数。如下例所示：

例:程序class\_test35

// C++虚函数表

#include <iostream>

class Base

{

public:

void func() // 不是虚函数，不会在虚函数表中

{

std::cout << "Base::func()" << std::endl;

}

virtual void f()

{

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

}

virtual void g()

{

std::cout << "Base::g()" << std::endl;

}

virtual void h()

{

std::cout << "Base::h()" << std::endl;

}

};

class Derive : public Base

{

public:

void f()

{

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

}

void g()

{

std::cout << "Derive::g()" << std::endl;

}

void h()

{

std::cout << "Derive::h()" << std::endl;

}

private:

virtual void test() // Derive类自有的虚函数，也会在Derive对象的虚函数表中

{

std::cout << "Derive::test()" << std::endl;

}

};

typedef void (\*Fun)(void); // 函数指针

int main(int argc, char\* argv[])

{

Base b;

Fun Fun\_ptr = NULL;

std::cout << "虚函数表地址：" << &b << std::endl;

int n = \*(int\*)(&b);

std::cout << (int\*)n << std::endl;

// 这里需要将&b强制转换成int\*(获取其十六进制地址)，然后取得其指向的对象（首地址，即第一个虚函数地址），然后在强制转换成int\*

// 第二次强制转换首先是为了获取十六进制的地址，其次是方便后面将其转换为Fun

// 指针可以相互强制转换

std::cout << "虚函数表-第一个函数地址：" << (int\*)\*(int\*)(&b) << std::endl;

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(&b));

Fun\_ptr(); // 调用Base::f()

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(&b) + 1);

Fun\_ptr(); // 调用Base::g()

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(&b) + 2);

Fun\_ptr(); // 调用Base::h()

Base\* b\_ptr = new Derive();

std::cout << "虚函数表地址：" << b\_ptr << std::endl;

std::cout << "虚函数表-第一个函数地址：" << (int\*)\*(int\*)(b\_ptr) << std::endl;

// Fun\_ptr = (Fun)\*((int\*)\*(int\*)(b\_ptr));

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(b\_ptr));

Fun\_ptr(); // 调用Derive::f()

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(b\_ptr) + 1);

Fun\_ptr(); // 调用Derive::g()

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(b\_ptr) + 2);

Fun\_ptr(); // 调用Derive::h()

// 这是虚函数表造成的问题，我们可以通过这种方式来访问派生类自有的

// 并且是私有的虚函数

Fun\_ptr = (Fun)\*((int\*)\*(int\*)(b\_ptr) + 3);

Fun\_ptr(); // 调用Derive::test()

delete b\_ptr;

b\_ptr = NULL;

return 0;

}

可以看出，在Base类型的对象b的内存中，存放了一个虚函数表，其中存放了Base类的3个虚函数的函数指针。而派生类Derive类的对象d的内存中，同样存放了一个虚函数表，由于在Derive类中重新实现了Base类的3个虚函数，所以对象d的虚函数表中，前3个函数指针替换成了Derive类的3个虚函数。这也算Base类的指针指向了Derive对象后，调用Base的虚函数，实际会调用Derive中的版本的原因所在。如下面两个内存图所示：

Base对象内存的虚函数表：





虚函数表造成的安全性问题：

我们可以通过上例中的方式来访问派生类自有的并且是私有的虚函数，如上例中的Derive中的test()函数。