Virtual Functions

C++通过virtual functions，从而使用dynamic binding实现多态性。In C++ dynamic binding happens when a virtual member function is called through a reference or a pointer to a base-class type.

1. **纯虚函数**才是不被实现的函数（只定义接口，起到该接口的规范作用，具体实现细节由各派生类决定，只要符合该规范即可）
2. **虚函数**中的`virtual`体现在“动态联编”或“动态绑定（延迟绑定）”上，一个类函数的调用不是在编译时刻决定的（普通函数，会直接在链接阶段，将函数地址链接到各个调用该函数的位置），虚函数则不能在编译时刻决定，如

   ```
   Base *a=new Derived(); // 指向基类的指针被绑定到派生类对象上，这个只有运行时才能决定
   a->func();  // 运行时，将Derived类中的override的virtual function，即func()的地址链接到调用位置
   ```

3.

> 一个和虚函数动态绑定的概念为**虚函数表**，可以帮助理解关于虚函数是如何实现动态绑定的。

如果不使用虚函数的话，则使用基类引用或者指针，调用相应的函数，则会被限制在基类函数本身，从而无法调用子类中被重写过的函数。

```
class Base {
public:
    void foo()const {
        std::cout << "Base foo()"<<std::endl;
    }
};
class Derived :public Base{
public:
    void foo() const {
        std::cout << "Derived foo()"<<std::endl;
    }
};
void fun(const Base& b) {
    b.foo();
}

Derived d;
Base b;
fun(d);
fun(b);
```

Output:

```
Base foo()
Base foo()
```

任何情况下，虚函数必须定义

一般来说，如果一个函数被声明，但是我们不使用它，那么它是可以不定义的。但是，虚函数的动态绑定特性决定了，编译器直到运行时才知道一个虚函数是否被使用（因为无法在运行时前基类的还是派生类的虚函数将会被调用），因此每个虚函数都要被定义。

基类和派生类的转换规则

1. 派生类到基类的转换只有通过指针或引用类型（引用实质也是一种指针，`*const`类型指针）相关的处理才会发生，如通过指针或引用调用虚函数`void fun(const Base &b)`。
   - 原因稍后再写
2. 基类到派生类的转换不可以是隐式的

> - there is no implicit conversion from the base-class type to the derived type.(显示拷贝构造或赋值，移动拷贝或移动赋值都可以)
>   其中，如果不是引用和指针，

```
void fun(const Base obj){
    //
}

Base obj1;
Derived obj2;

fun(obj1);// calls copy constructor defined in the base class to initialize parameter obj

fun(obj2);// equivalent to fun(Base(obj2));
```

Calls to Virtual Functions May be Resolved at Run Time

It is crucial to understand that dynamic binding happens only when a **virtual** function is called through **a pointer** or **a reference**.

When we call a virtual function on an expression that has a plain-nonreference and nonpointer-type, that call is bound at compile time.

> C++的多态性，通过引用或指针的静态类型和动态类型不同。也就是通过virtual function来实现动态绑定。
> 对于非虚函数而言，调用是在编译阶段绑定。
> 当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下，对象的动态类型才有可能与静态类型不同。

Virtual Functions in a Derived Class

When a derived class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains `virtual` in all the derived classes.

A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides.
`override`关键字就是为了防止基类和派生类的虚函数的参数列表不一致的情况。

```cpp
#include<iostream>
class Base {
public:
    void foo()const {
        std::cout << "Base foo()" << std::endl;
    }
};
class Derived :public Base{
public:
    // 这个版本会报错，即继承的虚函数重新定义和基类中的参数不一致
    void foo(int i) const override {
        std::cout << "Derived foo()"<<std::endl;
    }

    // 这个版本不会报错，但是没有体现多态性
    // 例如，下面的fun(const Base &)函数，不会调用foo(int i)，会调用基类的
    // foo()，因为派生类在基类的作用域内
    void foo(int i)const {
        //...
    }

    // 正确用法
    void foo(int i)const override{}
};

void fun(const Base& b) {
    b.foo();
}
```

The return type of a virtual in the derived class also must match the return type of the function from the base class. One exception applies to virtuals that return a reference or pointer to types that are themselves related by inheritance. That is, if `D` is derived from `B`, then a base class virtual can return a `B*` and the version in the derived can return a `D*`. However, such return types require that the derived-to-base conversion from `D` to `B` is accessible.

The `final` and `override` Specifiers

如果派生类定义了一个名字与基类中虚函数名字相同，但是形参不同的，这表明这两个函数是独立的。因此需要添加`override`关键字来指明该函数是重写的虚函数。

类似于final作用于类，即final类不可以被继承。一个函数也可以类似声明，即该函数不可以被override。

```
struct Base{
    void fun(int) const final;// 不允许后续的派生类覆盖fun(int)
};

struct Derived:public Base{
    void fun(int) const override;// error
};
```

Virtual Functions and Default Arguments

A virtual function can have default arguments. If a call uses a default argument, the value that is used is the one defined by the static type through which the function is called.

That is, when a call is made through a reference or pointer to base, the default argument(s) will be those defined in the base class. The base-class arguments will be used when the derived version of the function is run. In this case, the derived function will be passed the default arguments defined for the base-class version of the function. If the derived function relies on being passed different arguments, the program will not execute as expected.

> Virtual functions that have default arguments should use the same argument values in the base and derived classes.

Circumventing the Virtual Mechanism

In some cases, we want to prevent dynamic binding of a call to a virtual function; we want to force the call to use a particular version of that virtual. We can use the scope operator to do so. For example, this code:

```
double xxx=basep->Quote::net_price(42);
```

This call will be resolved at compile time.

> Warning: If a derived virtual function that intended to call its base-class version omits the scope operator, the call will be resolved at run time as a call to the derived version itself, resulting in an infinite recursion.

```
class Derived:public Base{
    void func(){
        Base::func();// calls the base-class func()
        func();// calls itself, resulting in an infinite recursion
    }
};
```