Class Scope under Inheritance

派生类的作用域嵌套在其基类的作用域之内。如果一个名字在派生类的作用域无法解析，则编译器在外层的基类作用域中寻找改名字的定义。

因此，派生类可以像使用自己成员一样使用基类的成员。（语义上，这些成员就是被继承过来的）

```
class Base{
    public:
    void func();
};
class Derived:public Base{
    // public inheritance
    // inherited func() remains to be public
};
Derived d;
d.func();
```

`func`函数解析过程如下：

- 用过`Derived`对象调用`func`，所以首先在`Derived`类中查找是否有次定义
- 接下来，在direct base class中查找
- 若仍然找不到，在indirect base class中查找

Name Lookup Happens at Compile Time

The staic type (non `static` member) of an object, reference, or pointer determines which members of that objects are visible. Even when the static and dynamic types might differ (as can happen when a reference or pointer to a base class is used), the static type determines what members can be used.

```
class Base{
public:
    int base_mem;
};

class Derived:public Base{
public:
    void foo();
    int derived_mem;
};

void func(Base &){
    //
}
Derived d;
Base b;
func(b);// 访问Base中的public成员
```

```
func(d);// 只可以访问d中的base-class部分的public成员


class DD:public Derived{
    public:
};

DD d;
DD *dptr=&d;// static and dynamic types are the same
Base *bptr=&d;// static and dynamic types differ
dptr->foo();// ok, dptr可以访问Derived中的foo()
bptr->foo();// error, bptr无权访问foo()
```

Derived中定义的foo对于bptr是不可见的。因为对于foo的定义查询是从Base开始的。然而Base中没有此成员。对于dptr，先从DD类内查找，若没有再从Derived中查找。再没有会从Base中查找。

Name Collisions and Inheritance

A derived class can reuse a name defined in one of its direct or indirect base classes. As usual, names defined in an inner scope (e.g., a derived class) hide uses of that name in the outer scope (e.g., a base class)"

```
struct Base{
    Base(const int &v=0):mem(v){}
    protected:
        int mem;
};
struct Derived: Base{
    // public inheritance by default

    Derived(const int &i=0):mem(i){}// initialize Derived::mem
    int get_mem()const{return mem;}// returns Derived::mem
    protected:
    int mem;    // hides mem in the base
};
```

类的作用域是内嵌在基类的作用域之内的，类比于其他类型的作用域，内部的作用域声明的成员会屏蔽外部的成员。

Using the Scope Operator to Use Hidden Members

Obviously, we can use a hidden base-class member by using the scope operator:

```
struct Derived:Base{
    // 前提是，该Base::mem对于派生类可访问
    int get_base_mem()const {return Base::mem;}
    // ...
};
```

> 除了override基类的虚函数之外，派生类最好不要重复使用基类中的名字。

Concepts as to name lookup and inheritance:
理解函数调用的解析过程可以帮助理解继承，对于`obj.mem()`来说：

- 首先确定`obj`的static type。因为我们调用的是一个成员，所以该类型必然是类类型。
- 在`obj`的static type对应的类中查找`mem`。如果找不到，则依次在直接基类中不断查找，直到到达继承链的顶端。若仍没有找到，则报错。
- 一旦找到`mem`，进行常规检查，对于当前找到的`mem`，本次调用是否合法。（比如，是否可以访问`mem`成员）
- 假设合法，则编译器根据调用的是否是虚函数而产生不同的代码：
    - 如果`mem`是虚函数且是通过指针或引用进行的调用，则编译器产生的代码将在run time确定运行该函数的哪一个版本，依据是该形参的动态类型
    - 反之，如果`mem`不是虚函数或者我们是通过对象进行的调用，则编译器将产一个常规函数调用的代码。

As Usual, Name Lookup Happens before Type Checking

Function declared in an inner scope do not overload functions declared in an outer scope. As a result, functions defined in a derived class do not overload members defined in its base class(es). If a member in a derived class (i.e., in an inner scope) has the same name as a base class member (i.e., a name defined in an outer scope), then the derived member hides the base-class member within the scope of the derived class. The base member is hidden ***even if the functions have different parameter list***:

```
struct Base{
    int memfcn();
};
struct Derived:Base{
    int memfcn(int);// hidders memfcn in the base
};
Derived d;
Base b;
b.memfcn(); // calls Base::memfcn
d.memfcn(10);    // callas Derived::memfcn
d.memfcn();      // error, memfcn with no arguments is hidden, that is
                 // d cannot access Base::memfcn
d.Base::memfcn();  // ok, calls Base::memfcn using scope operator
```

我们来看如何解析`d.memfcn()`，编译器会在`Derived`中查找名字为`memfcn`的成员定义。一旦找到，则不会继续扩大范围寻找（去基类中）。之后，检查是否合法。由于参数不匹配，因此发生错误。

> 编译器解析会先去搜索该名字的定义（按照一定规则），然后检查是否合法。

Virtual Functions and Scope

We can now understand why virtual functions must have the same parameter list in the base and derived classes. If the base and derived members took arguments that differed from one another, there would be no way to call the derived version through a reference or pointer to the base class. E.G.:

```
class Base{
    public:
        virtual int func();
};
class D1: public Base{
    public:

        // this would hide func in the Base; this func in not virtual
        int func(int); //parameter list differs from func in Base
        virtual void f2();// new virtual function that does not exist in Base
};
class D2:public D1{
    public:
        int func(int);// nonvirtual function hides D1::func(int)
        int func();// overrides virtual func from Base
        void f2();// overrides virtual f2 from D1
};
```

Calling a Hidden Virtual through the Base Class

Given the classes above, we give several different ways to call these functions:

```
Base bobj; D1 d1obj; D2 d2obj;
Base *bp1=&bobj, *bp2=&d1obj, *bp3=&d2obj;
bp1->func(); // virtual call, will call Base::func at run time
bp2->func(); // virtual call, will call Base::func at run time (因为D1类中没有
override基类Base中的虚函数func,因此直接继承基类的func,因为调用Base::func)
bp3->func(); // virtual call, will call D2::func at run time

D1 *d1p=&d1obj; D2 *d2p=&d2obj;
bp2->f2();  // error: Base has no member named f2

d1p->f2(); // virtual call, D1::f2() at run time
d2p->f2(); // virtual call, D2::f2() at run time
```

由于`func`是虚函数,且可以进行derived-to-base conversion。因而编译器会生成相应的代码,来决定运行时调用哪个版本的`func`。

对于非虚函数`func(int)`:

```
Base *p1=&d2obj; D1 *p2=&d2obj; D2 *p3=&d2obj;
p1->func(42); // error, Base has no version of func that takes an int

p2->func(42); // statically bound, calls D1::func(int)
p3->func(42); // statically bound, calls D2::func(int)
```

The dynamic type doesn't matter we call a nonvirtual function. The version that is called depends only on the static type of the pointer.

Overriding Overloaded Functions

overload:

```
void foo(int);
void foo(int, int);
```

override:

```
int mem;
int foo(int);
{
    int foo(int);
    int mem;// hide the definition outside
}

// override the virtual functions
class Base{
    public:
        void foo();
        void foo(int);
        virtual void func();
        virtual void func(int);
};
class Derived:public Base{
    public:
        void foo();
        void foo(int);
        void func();
        void func(int);
};
```

Member functions (virtual or otherwise) can be overloaded. A derived class can override zero or more instance of the overloaded functions it inherits. If a derived class wants to make all the overloaded versions available through its type, then it must override all of them or none of them.

Sometimes a class needs to override some, but not all, of the functions in an overloaded set. It would be tedious in such cases to have to override every base-class version in order to override the ones that the class needs to specialize.

```
#include <iostream>
class Base {
public:
        int foo() { return 1; }
```

```
        int foo(int v) { return 2; }
        int foo(int a, int b) { return 3; }

        virtual int bar()const { return 4;}
        virtual int bar(int a)const { return 5;}
        virtual int bar(int a, int b)const { return 6;}
};
class Derived :public Base {
public:
        //using Base::foo;
        int foo() { return 11;}
        //int foo(int a) { return 22; }
        //int foo(int a, int b) { return 33; }

        int bar()const { return 44; }
        int bar(int a)const { return 55; }
        int bar(int a, int b)const { return 66; }
};
int main() {
        Base b;
        Derived d;
        std::cout << d.foo(2);// call Derived::foo(int), which returns 22
}
```

❗ 为什么使用override部分重载函数，一般需要对基类中每个重载函数都需要重写？

如果我们不对每个重载函数都override，例如，以上代码只override了`foo()`一个，那么当我们调用`d.foo(2)`的时候，编译器解析，先去`Derived`类内查找此名字，找到后，发现只有`foo()`一个`foo`名字的函数，检查参数不匹配，因此会报错。

对于那些不希望override，希望直接继承的，则在`Derived`定义与基类中相同的即可。因此，若想通过派生类的对象访问所有的基类重载对象，则任何重载函数都不要在派生类中定义。此时，编译器解析会继续向其基类内查找名字定义。

我们可以使用`using`声明，来简化override的操作。`using`声明指定一个名字不指定形参列表，所以一条基类成员函数的`using`声明语句就可以把该函数的所有重载实例添加到派生类作用域中。此时，派生类只需要定义其特有的函数就可以了，而不用为继承而来的其他函数重新定义。

使用`using`后

```
class Derived :public Base {
public:
        using Base::foo;
        int foo() { return 11;}
    // other members as before
};
Derived d;
d.foo();// call Derived::foo()
d.foo(42);// call Base::foo(int), eqeuivalent to d.Base::foo(42).
d.foo(42,42);// call Base::foo(int)
```

这里可以和`swap`使用那里一样，使用`using`声明，

```
inline void swap(HasPtr &lhs,HasPtr &rhs){
    using std::swap;
    swap(lhs.ps,rhs.ps);
    swap(lhs.val,rhs.val);
}
```

编译器会解析会先从, inner scope查找`swap`名字, 查找不到, 则使用`std::swap`。