# **1 始终贯彻的基本原则 1.1 每个类负责定义各自的接口 1.1.1 每个类负责控制它自己的成员初始化（构造）以及析构的过程**

        每个类负责控制它自己的成员初始化以及析构的过程，因此派生类对象中包含的基类部分的成员也只能通过"递归"调用直接基类构造函数（直接基类构造函数可以在派生类构造函数的初始化列表中显式调用）和析构函数（因为析构函数不存在重载形式，所以C++标准规定只能由编译器在派生类的析构函数执行完后自动递归调用直接基类的析构函数）进行初始化和析构。

### 1.1.2 每个类负责控制它自己的成员访问权限

        C++通过成员访问说明符来控制类中所包含的各个成员在类的作用域的内部或外部的可访问性。 注：某个类成员是否能被访问取决于该语句所在的作用域，而与"是否通过对象或指针/引用来访问"无关。

|  |  |
| --- | --- |
| 123456789101112131415161718 | class Demo{public:    void f1(Demo &d)    {        d.f2(); //在Demo的作用域的内部，可以访问Demo的private成员    }private:    void f2()    {        cout << "void Demo::f2()" << endl;    }};int main(){    Demo d;    d.f1(d);    //在Demo的作用域的外部，可以访问Demo的public成员    d.f2();     //编译报错，在Demo的作用域的外部，不可以访问Demo的private成员return 0;} |

## 1.2 编译期间的通用名字查找规则和类专属名字查找规则

### 1.2.1 通用名字查找规则

|  |
| --- |
| 1. 首先，在名字所在的内层作用域中寻找其声明语句（包括类型声明、变量声明和函数声明）（且只考虑在该名字的使用之前出现的声明语句）；2. 如果没有找到，继续查找外层作用域（仍只考虑在该名字的使用之前出现的声明语句）；3. 如果最终没有找到匹配的声明，则程序报错；4. 若在上述步骤中的某个作用域中找到了所需的名字，则会立即结束名字查找的过程，编译器不会再到外层作用域中进行进一步的查找——由此带来了“内层作用域的名字会隐藏外层作用域同名的名字”的特性；5. 若在同一层作用域中出现两个以上的相同名字（即名字相同的不同声明语句），则编译报错（注：即使一个为变量名，另一个为函数名也会编译报错）。6. 除了在类内部出现的成员函数定义之外，在名字查找过程中总是向上查找作用域中的名字，由此带来“名字必须先声明后使用”的特性。 |

注1：“名字”指的是程序中使用到的名字（包括名称空间名、类型名、变量名和函数名等）；

注2：名字查找的本质：查找指定名字的声明语句；

注3：“名字查找”发生在“类型检查”之前——名字查找先于类型检查；

        “类型检查”包括如在赋值语句中右值变量的类型是否与左值变量相兼容等的变量类型检查、函数实参的类型是否和候选函数的形参类型相兼容的函数类型检查等。

注4：在同一个作用域中不能重复定义同一个名字，而在不同层次的作用域间则可以重复定义同一个名字；

注5：由于函数重载机制的存在，在同一个作用域中允许存在两个以上的名字相同但参数列表不同的函数声明；

### 1.2.2 类专属名字查找规则

       先使用通用名字查找规则编译类的所有成员声明，再编译类的成员函数定义体。

        因为成员函数体直到整个类可见后才会被处理，所以在成员函数体内可以使用已在类中任意位置定义的任何名字（如在某个成员函数体中，可以调用在后面才定义的成员函数）。成员函数体中使用的名字按照以下方式进行解析：

|  |
| --- |
| 1. 首先，在成员函数内查找该名字的声明——按照通用名字查找规则进行查找，即只有在该名字使用之前出现的声明才被考虑；2. 如果在成员函数内没有找到，则在类内继续查找，这时类的所有成员的声明都会被考虑；3. 如果类内也没有找到该名字的声明，再继续到类外部的作用域继续查找——按照通用名字查找规则进行查找，即在成员函数定义之前的作用域内继续查找。 |

|  |
| --- |
| int  bal;   //全局变量bal（全局作用域）int  val;class A{public:String getStr(void){ return  bal; }//bal为成员变量（先在块中找不到bal，再在类中就找到bal，就不需在全局作用域中找了）int  getVal(void){ return  val; }//val为全局变量（先后在块中和类中均找不到val，最后在全局作用域中找到val）    void  func(int  val);private:    string  bal; //成员变量bal（类作用域）};void  changeBal(string &bal);//全局函数changeBal()在类定义后，在被调用前声明void  A::func(int  val){    changeBal(bal); //先后在块中和类中都没有找到名字changeBal，最后在全局作用域中找到名字changeBal（注：函数changeBal()在调用前声明即可，不需要在类定义前就声明）} |
|  |

# 2 派生类和基类的继承关系

## 2.1 从代码构建的角度看继承关系——提高代码可复用性

  从代码实现的角度，若我们已经定义了一个类B，而类D需要包含和类B完全一样的接口，我们可以按照以下方式定义类D：

|  |
| --- |
| class  B{public:    void func1();    void setMem1(int mem1);    int  getMem1();private:    int  mem1\_;};class  D{public:    void func1();             //和在类B中定义的func1()函数完全一样    void func2();    void setMem1(int mem1);   //和在类B中定义的setMem1()函数完全一样    int  getMem1();           //和在类B中定义的getMem1()函数完全一样    void setMem2(int mem2);    int  getMem2();private:    int mem1\_;                //和在类B中定义的mem1\_变量完全一样    int mem2\_;}; |

  上述类D的定义的一个问题是我们分别在类B和类D中重复编写了func1()函数以及mem1\_变量的定义，代码复用率很低，而通过继承机制，我们可以通过使类D继承类B的方式，来避免在类D中编写重复的代码，如下所示：

|  |
| --- |
| class  B{public:    void func1();    void setMem1(int mem1);    int  getMem1();private:    int  mem1\_;};class  D : public B{public:    void func2();    void setMem2(int mem2);    int  getMem2();private:    int  mem2\_;}; |

## 2.2 从面向对象的角度看继承关系——建立类间从属逻辑关系

从面向对象程序设计的角度，通过继承联系在一起的若干个类之间构成一种（从属）层次关系，通常在层次关系的根部有一个基类（base class），其他类则直接或间接地从基类继承而来，这些继承得到的类成为派生类（derived class）。基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。

|  |
| --- |
| class Person{public:    int getAge() const;    void setAge(int value);    string getName() const;    void setName(const string &value);private:    int age\_;    string name\_;};class Student : public Person{public:    string getClassName() const;    void setClassName(const string &value);private:    string className\_;};class Teacher : public Person{public:    string subject() const;    void setSubject(const string &subject);private:    string subject\_;}; |

# 3 派生类（对象）

## 3.1 派生类对象的对象模型

一个派生类对象中包含以下两个组成部分：

|  |
| --- |
| 一个含有派生类自己定义的（非静态）成员的子对象；一个与该派生类继承的基类对应的子对象； |

 注1：类的静态成员不存放在对象中；

 注2：如果有多个基类（特指派生类直接继承了多个基类，即多继承的情况），那么基类子对象也有多个；

C++标准并没有明确规定派生类的对象在内存中如何分布（即没有规定派生类的对象的对象模型的实现方式），但是我们可以近似认为派生类对象的内存模型如下所示：

IMG_257

## 3.2 派生类对象的构造和拷贝控制过程（构造、赋值、析构）

    派生类对象中包含其所继承的基类部分子对象，也即包含其基类的所有数据成员，但因为C++标准规定每个类控制它自己的数据成员构造以及析构的过程，所以派生类无法直接控制它所继承的基类部分子对象中的数据成员的构造或拷贝控制过程，而必须通过"递归"地使用直接基类的构造函数或拷贝控制成员去初始化、赋值、销毁它的基类部分子对象。这就要求直接基类中对应的构造函数或拷贝控制成员对于派生类来说是可访问的，否则派生类就无法完成对应的构造或拷贝控制操作，而导致编译报错或合成删除的（=delete）操作。 所以构造函数不能是虚的？

### 3.2.1 派生类对象的构造过程

 派生类对象的直接基类部分与派生类对象自己的数据成员都是在派生类的构造函数的初始化阶段执行初始化操作的。类似于初始化数据成员的过程，派生类构造函数同样是通过构造函数初始化列表来将实参传递给直接基类构造函数，以实现“递归”地对其基类部分子对象的初始化，如下所示：

|  |
| --- |
| class Base{public:    //基类构造函数，用于初始化baseA\_和baseB\_    Base(int a, double b) : baseA\_(a), baseB\_(b)    {}private:    int baseA\_;    double baseB\_;};class Derived : public Base{public:    /\*        1. 在初始化列表中调用基类构造函数Base(a, b)来初始化派生类对象中基类部分的数据成员baseA\_和baseB\_；        2. 用实参c初始化派生类自己的数据成员derivedC\_    \*/    Derived(int a, double b, int c) : Base(a, b), derivedC\_(c)    {}private:    int derivedC\_;}; |

注1：除非我们在派生类构造函数的初始化列表中显式指出调用直接基类的哪个构造函数对其直接基类部分成员进行初始化，否则将调用直接基类的默认构造函数对此进行初始化（注：若此时直接基类不存在可访问的默认构造函数，则编译报错）。

注2：在派生类构造函数的初始化列表中只能调用其直接基类的构造函数对其直接基类部分的子对象进行初始化（注：若每一个派生类都负责调用其直接基类的构造函数对其直接基类部分子对象进行初始化，那么根据"递归"的思想，最终某个派生类对象的所有直接/间接基类部分子对象都能被初始化）。

3.2.1.1 派生类对象初始化的完整过程

|  |
| --- |
| 1. 按照直接基类的继承顺序，依次调用直接基类的构造函数来"递归"初始化相应直接基类部分的子对象；2. 按照派生类自己的数据成员的定义顺序，依次初始化各个数据成员；3. 执行派生类构造函数体中的语句； |

注1：适用于派生类同时继承多个直接基类的多重继承的情况

注2：直接基类部分子对象的初始化顺序只与继承的顺序有关，与在派生类构造函数的初始化列表中的调用顺序无关

注3：类数据成员的初始化顺序只与其声明的顺序有关与在类构造函数的初始化列表中的初始化顺序无关

### 3.2.2 派生类对象的拷贝和赋值过程

        和派生类的构造函数类似，派生类的拷贝/移动构造函数或拷贝/移动赋值运算符在拷贝/移动派生类自己的特有数据成员的同时，也要显式"递归"调用直接基类的拷贝/移动构造函数（注：在默认情况下，将隐式调用直接基类的默认构造函数来初始化直接基类部分的数据成员）或拷贝/移动赋值运算符来拷贝/移动其直接基类部分的成员。

3.2.2.1 派生类的拷贝/移动构造函数

    当为派生类定义拷贝/移动构造函数时，我们通常显式使用对应的直接基类的拷贝/移动构造函数初始化对象的直接基类部分，如下所示：

|  |
| --- |
| class Base{};class Derived : public Base{public:    //拷贝构造函数    Derived(const Derived &d) : Base(d) //显式调用直接基类的拷贝构造函数拷贝初始化直接基类部分的数据成员（注：派生类对象d可隐式转换为Base类型）    {}    //移动构造函数    Derived(Derived &&d) : Base(std::move(d)) //显式调用直接基类的移动构造函数移动初始化直接基类部分的数据成员（注：派生类对象d可隐式转换为Base类型）    {}}; |

3.2.2.2 派生类的拷贝/移动赋值运算符

        通常在派生类的拷贝/移动赋值运算符函数中，我们都会先通过调用直接基类的拷贝/赋值运算符函数来对直接基类部分的数据成员进行拷贝/移动后，再对派生类的特有成员进行拷贝/移动，如下所示：

|  |
| --- |
| class Base{};class Derived : public Base{public:    Derived &operator=(const Derived &rhs)    {        /\*在默认情况下，基类Base中的函数名字"operator="会被当前派生类Derived中的函数名字"operator="隐藏，因此，这里必须使用有限定的函数名字"Base::operator="，才会调用基类Base中的拷贝赋值运算符，否则，将递归调用Derived自己的拷贝赋值运算符        \*/        Base::operator=(rhs);   //显式调用直接基类的拷贝赋值运算符        //为派生类的特有数据成员赋值        //处理派生类特有成员的自赋值以及释放派生类特有的已有资源        //注：直接基类部分的成员的自赋值及资源释放，由Base::operator=(rhs)内部完成        return \*this;    }    Derived &operator=(Derived &&rhs)    {        /\*在默认情况下，基类Base中的函数名字"operator="会被当前派生类Derived中的函数名字"operator="隐藏，因此，这里必须使用有限定的函数名字"Base::operator="，才会调用基类Base中的移动赋值运算符，否则，将递归调用Derived自己的移动赋值运算符        \*/        Base::operator=(std::move(rhs));    //显式调用直接基类的移动赋值运算符        //注：移动赋值运算符不存在自赋值的问题        return \*this;    }}; |

### 3.2.3 派生类对象的析构过程

        派生类对象的析构过程与其初始化的过程类似，都需要"递归"地通过直接基类的析构函数来析构基类部分子对象，但是调用基类析构函数的过程，由编译器自动完成，因此派生类的析构函数只需负责销毁派生类自己分配的资源。最后需要注意的是，派生类对象的析构过程和其初始化的过程完全相反。

3.2.3.1 派生类对象析构的完整过程

|  |
| --- |
| 1. 执行派生类析构函数体中的语句；2. 按照派生类自己的数据成员的定义顺序，逆序依次析构各个数据成员；3. 按照直接基类的继承顺序，逆序依次析构各个直接基类部分的子对象； |

注1：由编译器自动调用各个数据成员所属类型的析构函数；

注2：由编译器自动调用各个直接基类的析构函数；

## 3.3 基类成员对派生类的可访问性

### 3.3.1 成员访问说明符

     C++通过成员访问说明符来控制类中所包含的各个成员在类的作用域的内部或外部的可访问性。派生类对象中的基类部分子对象属于基类作用域，而派生类部分子对象属于派生类作用域，因此派生类作用域属于基类作用域的外部（而不是内部）——即使派生类的名字作用域嵌套在其基类的名字作用域之内。

     C++为类成员提供三种成员访问说明符：public、protected和private，它们的功能如下表所示：

|  |  |
| --- | --- |
| 成员访问说明符 | 功能 |
| public | 在类的作用域的外部和内部均可访问 |
| protected | 在类的作用域内部以及在该类继承体系下的所有派生类作用域的内部均可访问，在其他作用域内均不可访问； |
| private | 仅能在类的作用域内部可访问，在类的作用域外部不可访问 |

### 3.3.2 派生访问说明符

        派生访问说明符是用来控制在派生类作用域的外部，对于派生类中直接基类部分成员的访问权限——可以看做用于控制直接基类中对派生类可见的成员（即直接基类中具有public和protected权限的成员），在派生类中应该具有怎样的成员访问说明符。

        C++为继承机制提供三种成员访问说明符：public、protected和private，它们的功能如下表所示：

|  |  |  |
| --- | --- | --- |
| 派生访问说明符 | 功能 | 备注 |
| public | 在派生类作用域的外部都可以访问派生类直接基类部分的public权限成员；而在当前派生类继承体系下的所有其他派生类作用域的内部（仍属于当前派生类作用域的外部）则还可以访问派生类直接基类部分的protected权限成员； | 相当于：直接基类中的public和protected成员在派生类中的权限仍然为public和protected不变； |
| protected | 在当前派生类继承体系下的所有派生类作用域的内部（仍属于派生类作用域的外部）可以访问派生类直接基类部分的public和protected权限成员；而在当前派生类作用域的其他外部则不可访问可以访问派生类直接基类部分的public和protected权限成员； | 相当于：直接基类中的public和protected成员在派生类中的权限都变成protected权限； |
| private | 在派生类作用域的外部都不可以访问派生类直接基类部分的成员； | 相当于：直接基类中的public和protected成员在派生类中的权限都变成private权限； |

### 3.3.3 类成员可访问性的本质——在编译时期的名字查找过程中，类成员声明的可见性

        成员访问说明符和派生访问说明符共同控制了类中指定成员在指定作用域下的可访问性，其本质是控制在指定作用域下进行名字查找时，指定类成员的声明在该作用域下的可见性。

## 3.4 继承体系的名字查找过程

### 3.4.1 在派生类作用域中声明的名字会隐藏在基类作用域中声明的同名名字

    虽然派生类的作用域嵌套在其基类的作用域之内，但是基类作用域属于派生类作用域的外层作用域，所以和其他内外层作用域关系一样，在派生类作用域中也能重新定义已在其外层基类（包括直接基类以及继承体系中的其他所有间接基类）作用域中定义的名字，此时定义在内层派生类作用域的名字将隐藏定义在外层基类作用域的同名名字。如下所示：

|  |
| --- |
| class Base{public:    Base() : mem\_(0){}    int mem\_;};class Derived : public Base{public:    int getMem(){        return mem\_;    //返回的是在派生类Derived作用域中的成员mem\_的值（Base::mem\_被隐藏了）    }    int mem\_;           //定义和外层基类Base作用域中成员Base::mem\_同名的成员};void test(){    Derived  d;    d.mem\_ = 32;        //给派生类对象d中的派生类子对象部分的成员mem\_赋值} |

注：除了覆盖继承而来的虚函数之外，派生类最好不要重定义已定义在其基类中的名字

3.4.1.1 需通过“基类::”访问基类中被隐藏的成员

C++标准允许我们通过作用域运算符显式指定编译器先到哪个作用域空间中查找指定的名字，做法是在需要访问的名字前添加"作用域名称::"前缀即可，如下所示：

|  |
| --- |
| class Base{public:void func(int a){cout << "Base::func(int a)" << endl}void func(){cout << "Base::func()" << endl; }    double mem\_ = 23.32;};class Derived : public Base{public:void func(){        //本质：this->Base::func();（因此"Base::func()"并不是静态函数的类名调用）        //注：编译器只会在Base作用域中查找名字"func"，若在Base作用域中查找不到名字"func"，不会再到Base的外层作用域中继续查找，而是直接编译报错        Base::func();   //输出：Base::func()        func();         //递归调用        cout << "Derived::func()" << endl;    }    int mem\_ = 32;};int main(){    Derived d1;    //error: no matching function for call to ‘Derived::func(int)’    d1.func(32);  //因为d1的静态类型为Derived，所以编译器从类Derived中开始进行名字查找 //显式指定编译器从类Derived中基类Base的作用域中开始进行名字查找（而不是从默认的Derived类中开始查找）    d1.Base::func(32);  //输出：Base::func(int a)    d1.func();          //输出：Derived::func()   //显式指定编译器从类Derived中Derived作用域中开始进行名字查找（而不是从默认的Derived类中开始查找）d1.Derived::func(); //输出：Derived::func()，等价于"d1.func();" //显式指定编译器从类Derived中基类Base的作用域中开始进行名字查找（而不是从默认的Derived类中开始查找）    d1.Base::func();    //输出：Base::func()    cout << d1.mem\_ << endl;           //输出：32//显式指定编译器从类Derived中Derived作用域中开始进行名字查找（而不是从默认的Derived类中开始查找）    cout << d1.Derived::mem\_ << endl;  //输出：32，等价于"cout << d1.mem\_ << endl;"//显式指定编译器从类Derived中基类Base的作用域中开始进行名字查找（而不是从默认的Derived类中开始查找）    cout << d1.Base::mem\_ << endl;     //输出：23.32    return 0;} |
|  |

### 3.4.2 通过作用域运算符显式指定使用基类中被隐藏的同名成员

|  |
| --- |
| class Base{public:    Base() : mem\_(0){}    int mem\_;};class Derived : public Base{public:    int getMem(){return Base::mem\_;  //返回的是在基类Base作用域中的成员mem\_的值}    int mem\_;               //定义和外层基类Base作用域中成员Base::mem\_同名的成员};void test(){    Derived  d;    d.Base::mem\_ = 32;      //给派生类对象d中的基类子对象部分的成员mem\_赋值} |

### 3.3.4 class和struct的区别

|  |
| --- |
| 1. class默认使用private作为类内部的成员访问说明符，而struct默认使用public；2. class默认使用private作为派生访问说明符去继承其他类，而struct默认使用public；3. 除上述两点区别外，class和struct没有任何其他区别。 |

## 3.5 对象调用语句的编译过程

### 3.5.1 知识铺垫：对象/表达式的静态类型和动态类型

|  |
| --- |
| 1.对象/表达式的静态类型表示对象声明时使用的类型或表达式所生成的类型，它在编译时期总是已知的；2.对象/表达式的动态类型则表示由对象或表达式表示的内存中的对象的类型，它直到运行时期才可知。3.如果对象/表达式既不是引用也不是指针，那么它的动态类型永远和静态类型一致；相反，若对象/表达式是引用或指针（特指基类引用或指针），则它的动态类型可能会与静态类型不一致。4. 一个类对象及其引用或指针的静态类型决定了该对象在编译时期哪些成员是可见的（即编译器是在静态类型所属的类中查找所需的成员），即使静态类型与动态类型可能不一致，但是我们能使用哪些成员仍然是由静态类型决定的。 |

### 3.5.2 对"p->mem()/p→mem（或robj.mem()/robj.mem）"的编译过程

|  |
| --- |
| 1.确定p（或者robj）的静态类型；（注：因为我们访问的是一个成员，所以该静态类型肯定是类类型）；2.在p（或者robj）的静态类型对应类的作用域中查找名字mem，如果找不到，则依次在外层直接基类作用域中不断查找，直至到达继承体系的顶端；如果找遍了该类及其所有基类仍然找不到，则编译报错；3. 一旦找到了名字mem，就进行常规的类型检查，以确定对于当前找到的名字"mem"是否合法；——类型检查4. 若"mem"为函数调用且调用合法，则编译器将根据调用的是否是虚函数来产生不同的代码：——“多态”的发生时间点4.1 如果"mem"是虚函数且我们是通过引用或指针进行的调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据的是对象的动态类型；4.2 反之，如果"mem"不是虚函数或者我们是通过对象（而不是引用或指针）进行的调用，则编译器将产生一个常规的函数调用。 |

注1：由上述过程可知，编译时名字查找的过程与“名字是变量名或函数名”以及“函数是否为虚函数”均无关。  
注2：成员虚函数只在生成二进制代码时和非虚函数有所区别，其他性质均和非虚函数完全一样。

## 3.6 派生类的声明

       派生类的声明和其他类的声明一样，声明中只需要包含类名，而不需要包含它的派生列表，如下所示：

|  |
| --- |
| class  Derived; //正确class  Derived : public Base;   //错误 |

## 3.7 阻止继承——final（C++11）

        有时我们会定义这样一种类：我们不希望其他类继承它，或者不想考虑它是否适合作为一个基类。为了实现这一目的，C++11标准提供了一种防止继承发生的方法，即在定义类时，在类名的后面添加关键字final，如下所示：

|  |
| --- |
| class NoDerived final{              //类NoDerived不能作为基类};class Base{};class Last final : public Base{     //继承自Base的类Last不能作为基类};class Bad1 : public NoDerived{      //编译报错：类NoDerived是final的};class Bad2 : public Last{           //编译报错：类Last是final的}; |

## 3.8 using“继承的”构造函数（C++11）

        在C++11标准中，派生类能够重用其直接基类定义的构造函数：可以让编译器为派生类生成与指定直接基类中除默认、拷贝和移动构造函数外的其他所有构造函数参数列表相同的构造函数。

### 3.8.1 使用using声明为派生类生成继承于直接基类的构造函数

        在通常情况下，using声明语句只是令某个名字在当前作用域内可见，而当using声明作用于直接基类的构造函数名时，using声明语句将令编译器产生代码。对于直接基类的每个构造函数（默认、拷贝和移动构造函数除外），编译器都会生成一个与之对应的派生类构造函数。也就是说，对于直接基类的每个构造函数（默认、拷贝和移动构造函数除外），编译器都在派生类中生成一个形参列表与之完全相同的构造函数。

        注：直接基类构造函数的explicit和constexpr属性可被派生类构造函数继承。编译器为派生类生成的继承于直接基类的构造函数形如：

|  |
| --- |
| // Derived：派生类的名字// Base：直接基类的名字// parms：构造函数的形参列表// args：将派生类的形参依次传递给直接基类构造函数的实参列表// 注：如果派生类含有自己的特有数据成员，则这些成员将被默认初始化Derived(parms) : Base(args){} |

|  |
| --- |
| class Base{public:    Base(int a, const string &s)        : a\_(a), s\_(s)    {}private:    int a\_;    string s\_;};class Derived : public Base{private: **//即使在private权限下，生成的继承于直接基类的构造函数仍是public的**    /\*        编译器为派生类生成如下所示的构造函数：        Derived(int a, const string &s):Base(a, s)        {}    \*/    using Base::Base;}; |

### 3.8.2 继承直接基类的using声明所属的成员访问说明符没有任何作用

        不管作用于直接基类构造函数名字的using声明出现在派生类中的哪个成员访问说明符下，都不会改变该直接基类构造函数原有的成员访问权限，而且生成的继承于该直接基类的构造函数的访问权限和该直接基类构造函数的权限相同，如下所示：

|  |
| --- |
| class Base{public:    Base(int a, const string &s)        : a\_(a), s\_(s)    {}private:    int a\_;    string s\_;};class Derived : public Base{private://即使在private权限下，生成的继承于直接基类的构造函数仍是和该直接基类构造函数的权限相同（public）    using Base::Base;};int main(){    //可以通过继承的构造函数创建派生类对象，可见该继承的构造函数是public的    Derived(32, "abc");    return 0;} |

|  |
| --- |
| class Base{protected:    Base(int a, const string &s)        : a\_(a), s\_(s)    {}private:    int a\_;    string s\_;};class Derived : public Base{private:    using Base::Base;};int main(){    //error: ‘Derived::Derived(int, const string&)’ is protected    Derived(32, "abc");    return 0;} |

### 3.8.3 当直接基类的构造函数的声明包含默认实参值时

        当一个直接基类的构造函数含有默认实参值时，这些默认实参值并不会被派生类构造函数继承。相反，派生类将从一个具有默认实参值的直接基类的构造函数中获得多个继承而来的构造函数，其中每个继承而来的构造函数分别省略掉最后的若干个含有默认实参的形参，如下所示：

|  |
| --- |
| class Base{public:    Base(int a=32, const string &s="abc", const vector<int> &v=vector<int>())        : a\_(a), s\_(s), v\_(v)    {}private:    int a\_;    string s\_;    vector<int> v\_;};class Derived : public Base{private:    //即使在private权限下，生成的继承于直接基类的构造函数仍是public的    /\*        编译器为派生类生成如下所示的构造函数：        Derived(int a):Base(a){}        Derived(int a, const string &s):Base(a, s){}        Derived(int a, const string &s, const vector<int> &v):Base(a, s, v){}        注：直接基类Base的默认构造函数Base()肯定不会被继承    \*/    using Base::Base;};int main(){    //调用派生类合成的默认构造函数Derived():Base(){}    auto d1 = Derived();    //调用派生类继承来的的默认构造函数    auto d2 = Derived(23);    auto d3 = Derived(23, "nba");    auto d4 = Derived(23, "nba", vector<int>{1,2,3});    //编译报错：invalid conversion from ‘const char\*’ to ‘int’    //因为派生类并没有从直接基类Base继承构造函数Derived(const string &s)    Derived("cba");    return 0;} |

# 4 多态（动态绑定）

## 4.1 派生类和基类（指针或引用）间的类型转换

### 4.1.1 派生类到基类（指针或引用）的隐式转换

如前文所述，在派生类对象中含有与其基类对应的组成部分，所以我们能把派生类的对象当成基类对象来使用，而且我们也能将基类的指针或引用绑定到派生类对象中的基类子对象部分上。

        编译器会隐式地执行派生类到基类的转换。这种隐式特性意味着我们可以把派生类对象或者派生类对象的引用用在需要基类引用的地方；同样地，我们也可以把派生类对象的指针用在用在需要基类指针的地方，如下所示：

|  |
| --- |
| void  test1(const Base &rb){}void  test2(const Base \*pb){}void  test(){    Dervied  d;    Dervied  &rd = d    test1(d);       //把派生类对象作为实参传递给基类引用形参    test1(rd);      //把派生类对象的引用作为实参传递给基类引用形参    test2(&d);      //把派生类对象的指针作为实参传递给基类指针形参} |

### 4.1.2 基类到派生类（指针或引用）的显式转换

        之所以存在派生类到基类的（隐式）类型转换，是因为每个派生类对象都包含一个基类部分子对象，而基类的引用或指针可以绑定到该基类部分子对象上。一个基类的对象既可以以独立的形式存在，也可以作为派生类对象的一部分存在。正因为一个基类的对象可能是派生类对象中的一部分，也可能不是，而编译器只能通过检查指针或引用的静态类型来推断某个转换是否合法，所以编译器在编译时无法确定某个特定的基类到派生类的转换在运行时是否安全，因此，在编译时不存在从基类向派生类的隐式类型转换。

        如果一个基类中含有一个或多个虚函数（使用dynamic\_cast类型转换运算符的前提，否则编译报错），我们可以使用dynamic\_cast类型转换运算符实现基类到派生类（或派生类到基类）的显式类型转换 将base 转换为derived，该转换的安全检查将在运行时执行（若该转换被判断为不安全，则dynamic\_cast表达式将返回nullptr或抛出bad\_cast异常），如下所示：

        注：dynamic\_cast类型转换运算符工作方式是通过源类型的虚函数表找到实例对象的RTTI信息的，而虚函数表的存在依赖于类是否定义了虚函数，因此基类到派生类的显式类型转换的前提基类中至少要含有一个虚函数，否则编译报错。

|  |
| --- |
| //基类不含有任何的虚函数struct Base{};struct Derived : Base{};int main(){    Derived d;    Base \*pb = &d;  //派生类到基类的隐式转换    //编译报错，error: cannot dynamic\_cast ‘pb’ (of type ‘struct Base\*’) to type ‘struct Derived\*’ (source type is not polymorphic)    Derived \*pd = dynamic\_cast<Derived \*>(pb);  //基类到派生类的隐式转换    return 0;} |

|  |
| --- |
| //基类包含一个虚函数（虚析构函数）struct Base{    virtual ~Base()=default;};struct Derived : Base{};int main(){    Derived d;    Base \*pb = &d;  //派生类到基类的隐式转换    //编译成功    Derived \*pd = dynamic\_cast<Derived \*>(pb);  //基类到派生类的隐式转换    return 0;} |

        另外，如果我们已知某个基类到派生类（或派生类到基类）的显式转换肯定是安全的，则还可以使用static\_cast来强制覆盖掉编译器在运行时的安全检查工作。

        最后需要注意的是，从语法角度，虽然C++支持使用dynamic\_cast和static\_cast对基类和派生类间的相互类型转换，但由于派生类到基类间的转换已存在隐式转换了（因为派生类到基类的转换绝对安全），所以在实践中通常只使用dynamic\_cast和static\_cast进行基类到派生类的显式类型转换。

### 4.1.3 不存在派生类和基类（对象）间的类型转换

        上述的基类和派生类间的（显式/隐式）类型转换都只对指针或引用类型有效，而在派生类对象和基类对象之间不存在这样的转换——即使在某些时候，C++标准允许出现形如将派生类对象转换为基类对象的语句，但实际上发生的并不是对象间的类型转换。

4.1.3.1 用派生类对象初始化/赋值给基类对象——派生类对象发生截断现象

        因为拷贝/移动构造函数和拷贝/移动赋值运算符的形参类型都是引用，所以派生类到基类的隐式转换允许我们给基类的拷贝/移动构造函数或拷贝/移动赋值运算符传递一个派生类的对象作为实参，也就是说，我们可以用一个派生类对象初始化/赋值给一个基类对象，但是因为此时调用的是基类的拷贝/移动构造函数或拷贝/移动赋值运算符，所以只会用到派生类对象实参中基类子对象部分的成员，而其派生类作用域中的特有成员将被忽略，我们称这种现象为：在使用派生类对象初始化/赋值基类对象的过程中，派生类对象发生了截断，如下所示：

|  |
| --- |
| class Base{public:    //合成的默认构造函数    Base()=default;    //合成的拷贝构造函数    Base(const Base &b)=default;    //合成的拷贝赋值运算符    Base &operator=(const Base &b)=default;public:    int b\_=23;};class Derived : public Base{public:    //合成的默认构造函数    Derived()=default;    //合成的拷贝构造函数    Derived(const Derived &b)=default;    //合成的拷贝赋值运算符    Derived &operator=(const Derived &b)=default;public:    int d\_=32;};int main(){    Derived d1, d2;    //本质调用Base的拷贝构造函数来初始化b1（而不是派生类对象到基类对象的类型转换）    Base b1 = d1;   //等价于： Base b1(d1);    //本质调用Base的拷贝赋值运算符来赋值b1（而不是派生类对象到基类对象的类型转换）    b1 = d2;        //等价于： b1.operator =(d2);    return 0;} |

4.1.3.2 用基类对象初始化/赋值给派生类对象——编译报错

        当我们使用基类对象初始化/赋值给派生类对象时，本质调用的是派生类对象的拷贝/移动构造函数或拷贝/移动赋值运算符，而它们的形参类型都是绑定到派生类的常量的引用，而因为不存在基类到派生类的隐式类型转换，因此编译报错，如下所示：

|  |
| --- |
| class Base{public:    //合成的默认构造函数    Base()=default;    //合成的拷贝构造函数    Base(const Base &b)=default;    //合成的拷贝赋值运算符    Base &operator=(const Base &b)=default;public:    int b\_=23;};class Derived : public Base{public:    //合成的默认构造函数    Derived()=default;    //合成的拷贝构造函数    Derived(const Derived &b)=default;    //合成的拷贝赋值运算符    Derived &operator=(const Derived &b)=default;public:    int d\_=32;};int main(){    Base b1, b2;    //本质调用Derived的拷贝构造函数来初始化d1    //error: conversion from ‘Base’ to non-scalar type ‘Derived’ requested    Derived d1 = b1;   //等价于： Derived d1(b1);    //本质调用Derived的拷贝赋值运算符来赋值d1    //error: no match for ‘operator=’ (operand types are ‘Derived’ and ‘Base’)    d1 = b2;        //等价于： d1.operator =(b2);    return 0;} |

引申：类的对象与其他任意类型对象之间的类型转换都是通过该类的成员函数完成的（转换构造函数和类型转换运算符）。

## 4.2 虚函数

        虚函数和非虚函数的唯一区别——当使用指针或引用调用虚函数时，则由编译器产生的代码将在运行时根据对象的动态类型再确定到底运行该虚函数的哪一个版本。

### 4.2.1 虚函数的限制

|  |
| --- |
| 构造函数不能是虚函数（注：析构函数、拷贝/移动赋值运算符及其他运算符重载函数都可以是虚函数）；原因：2. 静态成员函数不能是虚函数；原因：在编译时期就确定，是存储在静态存储区的3. 关键字virtual只能出现在类内部的声明语句之前，而不能用于类外部的函数定义；4. 如果基类把某个成员函数声明成虚函数，则该函数在派生类中的覆盖（重写）版本隐式地也是虚函数；5. 不管虚函数是否被调用到了，我们都必须为每一个虚函数都提供定义——因为编译器无法确定虚函数的哪个版本在运行时会被使用。原因：因为编译器编译的时候并不知道运行时会调用哪个版本， |

### 4.2.2 派生类覆盖基类虚函数

4.2.2.1 派生类覆盖虚函数的要求

如果需要在派生类中定义一个覆盖基类中某个虚函数的成员函数，则它必须满足以下条件：

|  |
| --- |
| 1. 它的函数名称和形参列表必须与被它覆盖的基类虚函数完全一致（注：包括this指针的const属性、noexcept异常说明等也需要保持一致）；2. 它的返回类型：（1）当基类的虚函数返回类型是基类本身的指针或引用时，且从派生类到基类的隐式类型转换是可访问的，那么派生类的覆盖版本的返回类型可以是派生类的指针或引用；（2）否则，派生类中的覆盖版本的返回类型也必须与基类相应的虚函数的返回类型完全一致。 |

注1：只有虚函数才存在“覆盖”一说；

注2：在派生类中也可以定义函数原型和基类中的某个非虚函数完全一样的特有成员函数（但这并不是对该基类非虚函数的“覆盖”），也不是重载，因为作用域不同，而是隐藏？因此在派生类中定义和基类中某个成员函数原型完全一样的特有成员函数并非只是虚函数的“权利”——因为基类和派生类中的this隐式形参类型是不一样的；

4.2.2.1.1 由编译器检查所覆盖的虚函数接口是否满足要求——override（C++11）

    如果在派生类中定义了一个与基类中某个虚函数的名字相同，但形参列表不同的成员函数，这仍然是合法的（注：在派生类作用域中进行名字查找时，会隐藏外层基类作用域中虚函数名字），但此时该定义在派生类中的成员函数显然不是基类中虚函数的覆盖版本。而在实践中，若我们希望在派生类中定义一个基类中某个虚函数的覆盖版本，但是因为手误等原因，使得派生类中定义的版本的参数列表与基类虚函数的参数列表不完全一致，但编译器却不会编译报错，且在通常情况下运行时也不会出错。

        在C++11标准中我们可以使用override关键字来显式说明派生类的某个函数是对基类虚函数的覆盖版本，这样编译器就会为我们检查在派生类中定义的版本是否满足覆盖基类中相应虚函数所需的一切条件，若不满足，则编译报错，如下所示：原因：让编译器帮忙检查格式，害怕你是想覆盖，但因为手误写错了

|  |
| --- |
| class Base{public:    virtual void func(const int &a) const {        cout << a << endl;    }};class Derived : public Base{public:    //使用override显式标记当前函数为基类某个虚函数的覆盖版本，让编译器进行检查    //error: ‘void Derived::func(int&)’ marked override, but does not override    void func(int &a) const override{        cout << a << endl;        a++;    }}; |

最后需要注意的是，因为override是为“覆盖”服务的，而“覆盖”只针对虚函数，所以override只能用于修饰虚函数，否则编译报错，如下所示：

|  |
| --- |
| class Base{public:    // func()为非虚函数    void func(const int &a){        cout << a << endl;    }};class Derived : public Base{public:    //使用override显式标记当前函数为基类某个虚函数的覆盖版本    //error: ‘void Derived::func(const int&)’ marked override, but does not override    void func(const int &a) override{        cout << a+1 << endl;    }}; |

4.2.2.2 阻止覆盖——final（C++11）

我们还能把基类的某个虚函数成员指定为final，则之后任何尝试覆盖该函数的操作都将导致编译报错，如下所示：

|  |
| --- |
| class Base{public:    //error: overriding final function ‘virtual void Base::func(const int&) const’    virtual void func(const int &a) const final{        cout << a << endl;    }};class Derived : public Base{public:    //error: virtual function ‘virtual void Derived::func(const int&) const’    void func(const int &a) const {        cout << a+1 << endl;    }}; |

 最后需要注意的是，和override一样，final是为“覆盖”服务的，而“覆盖”只针对虚函数，所以final也只能用于修饰虚函数，否则编译报错，如下所示：（可以修饰类，阻止类的继承）

|  |
| --- |
| class Base{public:    // func()为非虚函数    //error: ‘void Base::func(const int&)’ marked final, but is not virtual    void func(const int &a) const final{        cout << a << endl;    }};class Derived : public Base{public:    void func(const int &a) const {        cout << a+1 << endl;    }}; |

4.2.2.3 使用静态绑定调用虚函数的方法

4.2.2.3.1 通过域作用符显式指定所调用的虚函数的所在作用域

如前文所述，在派生类作用域中声明的名字会隐藏在基类作用域中声明的同名名字，而通过作用域运算符可显式指定使用基类中被隐藏的成员。所以，在派生类中定义的用于覆盖基类中虚函数的虚函数，也会隐藏基类中它所要覆盖的虚函数。但是存在这样的一种场景：基类的虚函数版本完成了继承体系中所有类型都要做的共同任务，而派生类重写的虚函数版本还需要执行一些与派生类本身密切相关的操作。在这种情况下，派生类的重写虚函数版本就应该先调用基类虚函数版本来完成一些共同任务，再执行自身类型相关的特有操作。但由于在派生类作用域中，基类的虚函数版本被隐藏了，所以需要通过作用域运算符来显式调用基类中的被隐藏的虚函数版本，如下所示：

|  |
| --- |
| class Base{public:    virtual void func(const int &a) const {        cout << a << endl;    }};class Derived : public Base{public:    void func(const int &a) const final override{        this->Base::func(a);    //显式指定调用基类版本的虚函数        cout << a+1 << endl;    }}; |

   在默认情况下，若通过基类指针/引用调用虚函数，编译器会为此生成动态绑定的可执行代码——根据运行时指针/引用的实际的动态类型调用相应版本的虚函数。再次通过作用域运算符，我们可在编译时显式指定编译器生成基类版本虚函数的可执行代码，而不是动态绑定的可执行代码，如下所示：

注：通过基类指针/引用只能通过作用域运算符显式调用基类中的虚函数，而不能显式调用派生类中的虚函数

|  |
| --- |
| class Base{public:    //非虚函数，静态绑定    void func(const int &a) const {        cout << a << endl;    }    //虚函数    virtual void virtualFunc() const{        cout << "void Base::virtualFunc()" << endl;    }};class Derived : public Base{public:    void func(const int &a){        cout << a+1 << endl;    }    void virtualFunc() const{        cout << "void Derived::virtualFunc()" << endl;    }};int main(){    int a = 23;    Derived d;    Base &rb = d;    //以下两个函数调用等价    //调用非虚函数，静态绑定，调用基类Base中的func()函数rb.func(a);              //输出：23    //调用非虚函数，静态绑定，显式指定调用基类Base中的func()函数    rb.Base::func(a);           //输出：23    //以下两个函数调用不等价    //调用虚函数，动态绑定，可能调用基类Base中的虚函数，也可能调用派生类Derived中的虚函数    rb.virtualFunc();    //输出：void Derived::virtualFunc()//显式指定调用基类Base中的虚函数，静态绑定    rb.Base::virtualFunc(); //输出：void Base::virtualFunc()    //因为rb的静态类型为基类Base，而在Base作用域中并不包含名为Derived的作用域，故编译报错    //error: ‘Derived’ is not a base of ‘Base’    rb.Derived::virtualFunc();    return 0;} |

4.2.2.3.2 在构造函数和析构函数中调用虚函数

        在构造函数和析构函数体中，this对象可能处于不完整的状态（如类的构造函数可能被其派生类对象的构造函数调用，此时派生类部分子对象还未开始构造；类的析构函数可能在其派生类对象的析构函数之后被调用，此时派生类部分子对象已经被销毁），所以编译器规定在构造函数体或析构函数体内所调用的虚函数都使用静态绑定。

## 4.3 虚析构函数

        当我们使用delete运算符作用于一个类指针时，便会开始销毁该指针所指向的堆对象的流程，期间会调用该类的析构函数，以释放该类对象中存在的堆资源。若指针是派生类类型的指针，那么在释放完自身特有成员后，将继续依次调用其直接基类的析构函数，以对其直接基类部分子对象进行释放，以此类推，直至将继承体系上的所有子对象释放完成，如下所示：

|  |
| --- |
| class Base{public:    Base():pb\_(new int(23)){}    ~Base(){        //基类析构函数负责释放基类作用域中的堆资源        if(pb\_) delete pb\_;    }private:    int \*pb\_;};class Derived : public Base{public:    Derived():pd\_(new int(32)){}    ~Derived(){        //派生类析构函数负责释放派生类作用域中的堆资源        if(pd\_) delete pd\_;    }private:    int \*pd\_;};class Derived\_Derived : public Derived{public:    Derived\_Derived():pdd\_(new int(17)){}    ~Derived\_Derived(){        //派生类析构函数负责释放派生类作用域中的堆资源        if(pdd\_) delete pdd\_;    }private:    int \*pdd\_;};int main(){    Derived \*pd = new Derived\_Derived;    //默认情况下，只会依次调用析构函数~Derived()和~Base()，而不会调用派生类的析构函数~Derived\_Derived()    //从而导致派生类中的堆资源Derived\_Derived::pdd\_得不到释放，而产生内存泄露    delete pd;    return 0;} |

        正如通过虚函数来实现动态绑定一样（根据在运行时调用虚函数的动态类型来确定先调用哪个版本的虚函数），我们可以通过virtual关键字将析构函数定义为虚析构函数，从而实现对象析构的动态绑定：若基类指针指向的对象是派生类对象，那么先调用派生类的析构函数（后续会根据析构的完整过程依次调用基类的析构函数）；若基类指针指向的对象是基类对象，那么先调用基类的析构函数。如下所示：

|  |
| --- |
| class Base{public:    Base():pb\_(new int(23)){}    //虚析构函数    virtual ~Base(){        //基类析构函数负责释放基类作用域中的堆资源        if(pb\_) delete pb\_;    }private:    int \*pb\_;};class Derived : public Base{public:    Derived():pd\_(new int(32)){}    //和其他虚函数一样，若基类的析构函数为virtual，那么派生类的析构函数默认也是virtual的    ~Derived(){        //派生类析构函数负责释放派生类作用域中的堆资源        if(pd\_) delete pd\_;    }private:    int \*pd\_;};class Derived\_Derived : public Derived{public:    Derived\_Derived():pdd\_(new int(17)){}    //和其他虚函数一样，若基类的析构函数为virtual，那么派生类的析构函数默认也是virtual的    ~Derived\_Derived(){        //派生类析构函数负责释放派生类作用域中的堆资源        if(pdd\_) delete pdd\_;    }private:    int \*pdd\_;};int main(){    Derived \*pd = new Derived\_Derived;    //因为基类指针指向的是派生类对象，所以先调用派生类的析构函数~Derived\_Derived()释放派生类作用域中的堆资源Derived::pdd\_，    //再调用基类的析构函数~Derived()和～Base()释放基类作用域中的对资源Derived::pd\_和Base::pb\_，因此不会发生内存泄露    delete pd;    return 0;} |

## 4.4 抽象类

### 4.4.1 纯虚函数

（1）为某个类定义纯虚函数的目的：不希望用户创建该类对象；该类的设计无法确定该虚函数的默认实现，需强制其派生类进行覆盖实现（即通过接口的方式强制派生类保持接口的一致性）。接口

（2）一个纯虚函数无须定义；只声明，不能定义对象

（3）通过在类的内部成员函数声明语句的分号之前书写=0来将一个虚函数声明为一个纯虚函数；

（4）=0只能出现在类内部的虚函数声明语句中，而不能出现在类外部；

### 4.4.2 抽象类——存在纯虚成员函数的类

含有纯虚函数或未经覆盖从基类直接继承纯虚函数的类是抽象类，在C++中抽象基类负责定义接口，而后续继承抽象基类的其他派生类可以覆盖该接口（即实现接口）。因为抽象类中包含未被实现的纯虚函数声明，所以我们不能创建抽象类的对象。即便这样，因为每个类负责定义各自的接口，所以抽象基类的派生类仍需要通过该抽象基类的构造函数/析构函数来初始化/释放该抽象基类部分子对象的数据成员，因此我们仍需要为抽象基类定义构造函数和析构函数。

需要注意的是，在抽象类中，还是可以像普通类一样定义自己的数据成员和函数成员。

# 5 多继承

   多重继承是指从多个直接基类中产生派生类的能力，多重继承的派生类继承了所有基类的属性。对于派生类能够继承的直接基类个数，C++标准没有进行特殊规定，但是在某个给定的派生列表中，同一个直接基类只能出现一次。

## 5.1 名字查找结果的二义性

        在单继承的情况下，派生类的作用域嵌套在直接基类和间接基类的作用域中，在名字查找的过程中沿着继承体系自底向上进行，直到找到所需的名字，由此使得派生类作用域中的名字将隐藏基类作用域的同名成员。在多继承的情况下，和上述相同的名字查找过程递归地在所有直接基类中同时（并行）进行，如果同一个名字在多个直接基类中被递归找到，则对该名字的使用将具有二义性，导致编译报错。

### 5.1.1 变量名字的二义性

        假设派生类Derived同时继承了基类Base1和Base2，如果我们通过Derived的对象、指针或引用使用了某个名字，则编译器会并行地在由Base1和Base2两个不同继承分支确定的两棵子树上查找该名字，如果名字在超过一棵子树中被找到，则该名字的使用具有二义性。

        注：即使该名字在一个基类中是私有的，而在另一个基类中是公有的或是受保护的，同样会发生二义性错误；

### 5.1.2 函数名字的二义性

        在多继承的场景下，因为位于不同继承分支上的两个基类作用域属于不同的两个作用域，所以对于这两个作用域中的同名函数不存在函数重载机制（函数重载要求能相互重载的若干个同名函数必须位于同一个作用域中），因此，若派生类分别从来自不同继承分支的两个基类中继承了两个同名函数（即使它们的参数列表不尽相同），若使用派生类对象且不加前缀地调用该函数，也会导致二义性错误。

### 5.1.3 解决方案

        对于一个派生类来说，从它的几个基类中分别继承相同的成员是安全合法的，只是在使用的过程中会出现潜在的编译二义性问题。可通过以下两种方法之一来避免这种二义性的发生：

|  |
| --- |
| 1. 在使用某个名字时通过域作用符显式指定该名字所在的基类作用域作用域（通过显式指定编译器名字查找的起始作用域来避免二义性错误）2. 在派生类中为某个名字定义一个新版本（利用“派生类作用域中的名字会隐藏基类作用域中的同名名字”的机制来避免二义性错误） |

## 5.2 派生类对象中包含多个相同的基类子对象——间接导致名字查找结果的二义性

        尽管在派生列表中同一个基类只能出现一次，但实际上派生类仍可能多次继承同一个基类：派生类可以通过它的两个直接基类分别继承同一个间接基类，也可以直接继承某个基类，然后通过另一个基类再一次间接继承该基类，如下所示：

在默认情况下（即在非虚继承的情况下），派生类中含有继承链上每个基类对应的子部分，因此，如果某个类在派生过程中出现了多次，则派生类中将包含该类的多个子对象（因为在派生类构造函数初始值列表中是相互独立“递归”地调用直接基类）。

### 5.2.1 解决方案——虚继承

        在C++语言中我们通过虚继承的机制解决上述的问题。虚继承的目的是令某个类在继承基类时做出声明，承诺愿意共享它所继承的虚基类。其中，可被共享的基类子对象称为虚基类。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。我们通过在派生列表中添加关键字virtual来指定需要继承的虚基类，如下所示：

|  |
| --- |
| //注：关键字public和virtual的顺序随意class istream : public virtual ios{};class ostream : public virtual ios{};class iostream : public istream, public ostream{}; |

   virtual说明符表明了一种愿望，即在后续的派生类当作共享虚基类的同一份实例。如上述代码实例中，istream和ostream都使用了虚继承方式继承ios，表示它们都愿意后续的派生类共享同一份ios子对象实例，这样iostream才能使用一份ios实例；若istream和ostream中有一个使用非虚继承方式继承ios，不妨假设istream使用虚继承，而ostream使用非虚继承，那么由于ostream不允许其后续的派生类共享其直接基类ios的实例，iostream中仍会存在两份ios子对象实例（即使istream允许共享ios实例）

# 6 注意事项

## 6.1 通过“基类::”访问基类中被隐藏的成员时，编译器只会在基类作用域中进行名字查找

|  |
| --- |
| void func(){    cout << "::func()" << endl;}class Base{public:    void func(){        cout << "Base::func()" << endl;    }};class Derived : public Base{public:    void func(){        //本质：this->Base::func();（因此"Base::func()"并不是静态函数的类名调用）        //注：编译器只会在Base作用域中查找名字"func"，若在Base作用域中查找不到名字"func"，不会再到Base的外层作用域中继续查找，而是直接编译报错        Base::func();   //输出：Base::func()        cout << "Derived::func()" << endl;    }}; |

|  |
| --- |
| void func(){    cout << "::func()" << endl;}class Base{};class Derived : public Base{public:    void func(){        //本质：this->Base::func();（因此"Base::func()"并不是静态函数的类名调用）        //注：编译器只会在Base作用域中查找名字"func"，若在Base作用域中查找不到名字"func"，不会再到Base的外层作用域中继续查找，而是直接编译报错        Base::func();   //编译报错，error: ‘func’ is not a member of ‘Base’        cout << "Derived::func()" << endl;    }}; |

## 6.2 在默认情况下基类和派生类的同名非静态成员函数不会重载

     重载函数的定义：如果在同一作用域内有若干个函数名字相同，但形参列表不同的函数声明（注：只涉及到声明，而不涉及到定义），我们称它们为重载（overload）函数。

    因为派生类中的派生类作用域属于内层作用域，而其基类作用域属于外层作用域，二者不属于同一个作用域，所以声明在派生类作用域中的成员函数不会重载声明在其基类作用域中同名的成员函数——表现出隐藏基类中同名的成员函数的特性。

|  |
| --- |
| class Base{public:    void func(int a){        cout << "Base::func(int a)" << endl;    }};class Derived : public Base{public:    void func(){        cout << "Derived::func()" << endl;    }};int main(){    Derived d1;    //编译报错：no matching function for call to ‘Derived::func(int)’    d1.func(32);  //因为d1的静态类型为Derived，所以编译器从类Derived中开始进行名字查找    d1.func();   //输出：Derived::func()    return 0;} |

### 6.2.1 实现派生类和基类中同名的非静态成员函数进行重载的方法

因为函数重载的前提是名字查找（即在确定候选函数阶段找到所有指定的函数名字），而在默认情况下，编译器一旦在内层派生类作用域中找到了指定名字的函数声明，那么就不会到外层基类作用域中继续查找了，因此在基类中定义的成员函数并不能与在派生类中定义的同名成员函数构成重载（即函数重载只发生在同一个作用域中）。

        但是在派生类中通过using声明就可以将在基类中的同名函数的声明引入到派生类作用域中，这样编译器在派生类作用域中也可以找到声明在基类作用域中的名字，从而使得基类和派生类的同名函数能相互重载，如下所示：只用引入声明就可以

|  |
| --- |
| class Base{protected:    void func(int a){        cout << "Base::func(int a)" << endl;    }};class Derived : public Base{};class Derived\_Derived : public Derived{public:   //引入基类成员函数名称，使得基类中的" void func(int a)"和派生类中的" void func()"相互重载using Base::func;  void func(){        cout << "Derived\_Derived::func()" << endl;    }};int main(){    Derived\_Derived dd;    dd.func();      //输出：Derived\_Derived::func()    dd.func(32);    //输出：Base::func(int a)    return 0;} |