组合与继承

组合与继承

By 曹菡雯（计 03）、赵晨阳（计 06）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分难度不大，并没有太多扩展内容。主要是 L7-组合与继承的课堂笔记整理，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于每一页 PPT 都有完整的解读，同时联系了前后课程的内容与课程作业，对于课件中的操作也有一定的扩展。

本次笔记的代码部分来自于 Vim 编辑器，从之前笔记的纯粹黑白色调改为了彩色调，有助于阅读。

如果阅读时间不够充足，建议阅读课堂的扩展部分。

1.1.3 public 接口访问 private 数据

3.3 重写覆盖的总结

一、组合

1.1 定义

如果对象 a 是对象 b 的一个组成部分，则称 b 为 a 的整体对象，a 为 b 的部分对象。并把 b 和 a 之间的关系，称为“整体—部分”关系（也可称为“组合”或“has-a”关系）。

1.2 两种实现方式

1.2.1 公有成员

已有类的对象作为新类的公有数据成员，这样通过允许直接访问子对象而“提供”旧类接口。

1.2.2 私有成员

已有类的对象作为新类的私有数据成员。新类可以调整旧类的对外接口，可以不使用旧类原有的接口（相当于对接口作了转换）

我们如下命名两类组合方式：

```
private:
  Wheel w;//private 组合
public:
  Engine e; //public 组合
```

1.2.3 public 接口访问 private 数据

```
#include <iostream>
```

```
using namespace std;
class Wheel{
   int _num;
public:
   void set(int n){_num=n;}
};
class Engine{
public:
  int _num;
  void set(int n){_num=n;}
};
class Car{
private:
  Wheel w;
public:
  Engine e; /// 公有成员，直接访问其接口
  void setWheel(int n){w.set(n);} /// 提供私有成员的访问接口
};
int main()
{
  Car c;
  c.e.set(1);
  c.setWheel(4);
  return 0;
}
```

Car 由 Engine 和 Wheel 两部分组成。如果我的 Wheel 设为私有，那么我没法直接在类外（主函数里）访问 w.set(n)，因为类外无法访问私有成员。但是我可以设置 public 接口 setwheel，setweel 是个 public 的类内函数，可以访问类内 private 成员，从而我能够通过此接口访问到 private 数据成员 wheel w。同理，观察 wheel 类和 Engine 类的写法，其实也是这种给 private 对象添加 public 接口的思想。

将此例子进一步阐述

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

class Wheel{
   int _num;
public:
   Wheel(int x){_num=x;}
   void set(int n){_num=n;}//没有 get 函数是没法直接在 Car 类里访问到 private

成员的 private 的
```

```
    int getwheelvalue(){return this->_num;}
};
class Engine{
public:
    Engine(int y){_num=y;}
    int _num;
    void set(int n){_num=n;}
};
class Car{
private:
  Wheel w;
public:
  Engine e; /// 公有成员，直接访问其接口

  void setWheel(int n){w.set(n);} /// 提供私有成员的访问接口

  Car(int x,int y):w(x),e(y){};
  //Car(int x):Egine(x),Wheel(x){};
  void print(){cout<<"my Engine is "<<this->e._num<<" my Wheel is "<<this->w.get
wheelvalue()<<endl;};
};
int main()
{
  Car c(5,3);
  c.print();
  c.e.set(1);
    c.print();
  c.setWheel(4);
    c.print();
  return 0;
}
Output:
my Engine is 3 my Wheel is 5
my Engine is 1 my Wheel is 5
my Engine is 1 my Wheel is 4
```

第一，Wheel w 是 Car 的私有成员，Car 的 public 接口可以访问 Car 的 private 数据成员，故而可以访问 w，但是无法访问私有数据成员的私有数据成员，故而无法直接在 print 中输出 w._num，我们选择了再对 Wheel 设计私有数据成员的接口 getwheelvalue 来访问 wheel 的_num。

第二，对于 Car 的构造函数的初始化列表，当 Car 的数据成员是 int x 时，我们写的是 x(i)，而不是 int(i)。（否则，如果有多个 int 类的成员，初始化就无法进行了）所以这里写的也是 w(x)，而不是 Wheel(x)。这里其实也体现了基本数据和自定义类的类比关系。

第三、我们这里采用了初始化列表来构造 Car，而不是采用函数体内赋值。因为 C++11 直接禁止了函数体内赋值。

我们对此的理解：如果是函数体内赋值来构造 Car，那么需要定义 Wheel 和 Engine 的默认构造函数（如果我们不定义带有参数的构造函数，系统会自动生成）除此之外，类似 w=5 这样的语句还需要隐式调用带有参数的构造函数生成 Wheel（5）,然后调用隐式生成的移动赋值语句给 w 赋值。这个方式比起初始化列表效率低下太多。（如此禁止可能还有其他的因果关系）

这其实就是 1.3.1 的第一句话的理解。

1.3 构造与析构

1.3.1 子对象参数构造

子对象构造时若需要参数,则应在当前类的构造函数的初始化列表中进行。不能够通过函数体内赋值。

具体而言,C++11 的规定是其他类的子对象必须要通过初始化列表来构造，如果不写在初始化列表里，那么就调用默认构造。

如果 A 类内组合有其他类 B 的对象，那么 B 的对象必须要通过初始化列表来构造，可以不写初始化列表，那么会调用 B 的默认构造函数。但是一定不能在 A 的构造函数体内进行构造，效率太低。

这个地方感觉会和我们的理解发生冲突，因为类当中经常会需要 int 数据成员。int 是一个类，那难道 int 只能在初始化列表里赋值吗？

这个理解是错的。int 不能算是一个类，它和 char,long,double 都只是基本的变量类型。组合指的是类和类之间，数据成员里的 int，这不是组合。基本数据类型和自己写的类不是一个东西。而且对于短赋值语句，尽量能放初始化列表就不放函数体内，前者效率更高。

但是 Vector 是个封装好了的类，#include <vector> using std::vector，就是把包含它的库搞到你的程序中。

1.3.2 子对象默认构造

若使用默认构造函数来构造子对象，则不用做任何处理。

1.3.3 构造次序

先完成子对象构造，再完成当前对象构造。

1.3.4 析构次序

子对象构造的次序仅由在类中声明的次序所决定。

析构函数的次序与构造函数相反，也就是完全反着析构一次即可。

1.3.5 例子

例一 构造与析构的执行顺序

```
#include <iostream>
```

```
using namespace std;
class S1 { //Single1 类别
  int ID;
public:
  S1(int id) : ID(id) { cout << "S1(int)" << endl; }
  ~S1() { cout << "~S1()" << endl; }
};
class S2 {//Single2 类别
public:
  S2() { cout << "S2()" << endl; }
  ~S2() { cout << "~S2()" << endl; }
};
class C3 {//Composite3 类别
  int num;
  S1 sub_obj1; /// 构造函数带参数
  S2 sub_obj2; /// 构造函数不带参数
public:
  C3() : num(0), sub_obj1(123) /// 构造函数初始化列表中构造子对象
        { cout << "C3()" << endl; }
  C3(int n) : num(n), sub_obj1(123)
        { cout << "C3(int)" << endl; }
  C3(int n, int k) : num(n), sub_obj1(k)
        { cout << "C3(int, int)" << endl; }
  ~C3() { cout << "~C3()" << endl; }
};
int main()
{
  C3 a, b(1), c(2), d(3, 4);
  return 0;
}
S1(int)
S2()
C3()
S1(int)
S2()
C3(int)
S1(int)
S2()
C3(int)
S1(int)
S2()
C3(int, int)
~C3()
~S2()
```

```
~S1()
~C3()
~S2()
~S1()
~C3()
~S2()
~S1()
~C3()
~S2()
~S1()
```

输出结果，前面 12 行，每三行为一个单位，是一个构造。且先按照声明次序构造子对象 sub_obj1，之后是 sub_obj2，最后完成 C3 构造。

至于析构顺序，那就纯粹是把前 12 行向下对折。

例二 对象组合的拷贝与赋值（对课件上的例子进行优化，更清楚些）

```
#include <iostream>
using namespace std;
class C1{
public:
   int i;
   C1(int n):i(n){cout<<"C1 with int: "<<n<<endl;}
   C1(const C1 &other) /// 显式定义拷贝构造函数
      {i=other.i; cout << "C1(const C1 &other)" << endl;}
};

class C2{
public:
   int j;
   C2(int n):j(n){cout<<"C2 with int: "<<n<<endl;}
   C2& operator= (const C2& right){/// 显式定义赋值运算符
      if(this != &right){
         j = right.j;
         cout << "operator=(const C2&)" << endl;
      }
      return *this;
   }
};
class C3{
public:
   C1 c1;
   C2 c2;
   C3():c1(0), c2(0){cout<<"C3 without arg"<<endl;}
```

```
    C3(int i, int j):c1(i), c2(j){cout<<"C3 with i and j"<<endl;}
    void print(){cout << "c1.i = " << c1.i << " c2.j = " << c2.j << endl;}
};
int main(){
    C3 a(1, 2);
    C3 b(a);  //C1 执行显式定义的拷贝构造，C2 执行隐式定义的拷贝构造
    cout << "b: ";
    b.print();
    C3 c;
    cout << "c: ";
    c.print();
    c = a;  //C1 执行隐式定义的拷贝赋值，C2 执行显式定义的拷贝赋值
    cout << "c: ";
    c.print();
    return 0;
}
Output：
C1 with int: 1
C2 with int: 2
C3 with i and j
C1(const C1 &other)
b: c1.i = 1 c2.j = 2
C1 with int: 0
C2 with int: 0
C3 without arg
c: c1.i = 0 c2.j = 0
operator=(const C2&)
c: c1.i = 1 c2.j = 2
```

例三、区分数据来源

```cpp
#include <iostream>
using namespace std;
class A {
  int data;
public:
   A():data(0)
    {cout << "A::A(" << data << ")\n";}
   A(int i):data(i)
    {cout << "A::A(" << i << ")\n";}
  };
class B {
  int data{2018};
  A a;
public:
  B(){}
  B(int i):a(i){}
  void print()
     {cout << "data = " << data << endl;}
};
int main() {
  B obj1;
  B obj2(2019);
  obj1.print();
  obj2.print();
  return 0;
}
```

A B类的默认构造函数没有显式调用A类的构造函数，此时编译器会自动调用A类的默认构造函数

B B类的普通构造函数可以在初始化列表中显式调用A类的普通构造函数

C 该程序的输出为 A::A(0)\nA::A(2019)\ndata = 2018\ndata = 2019\n

D obj1析构时先执行B类的析构函数，再执行A类的析构函数

提交

16

注意，int data{2018};和 int data=2018;这两个写法一致。这里 B 类的 print 函数输出的是 B 的 data，而不是数据成员 a 的 data。但是构造函数 B(int i)a(i)是用 2019 在构造 a，这一个小细节比较坑。

二、继承

2.1 定义

一般—特殊”结构，也称“分类结构”，是由一组具有“一般—特殊”关系的类所组成的结构，C++使用继承来表达类间的“一般—特殊结构”。

A 继承 B，则：

属性和服务上：类 A 具有类 B 全部的属性和服务，而且具有自己特有的某些属性或服务。A 为 B 的特殊类，B 为 A 的一般类。

对象关系上：类 A 的全部对象都是类 B 的对象，而且类 B 中存在不属于类 A 的对象。A 是 B 的特殊类，B 是 A 的一般类。

2.2 继承方式与语法

被继承的已有类，被称为基类(base class)，也称“父类”、“一般类”。

通过继承得到的新类，被为派生类(derived class，也称“子类”、“扩展类”、“特殊类”。

常见的继承方式：public, private

class Derived : [private] Base { .. }; 缺省继承方式为 private 继承。

class Derived : public Base { ... };

protected 继承很少被使用

class Derived : protected Base { ... };

2.3 无法继承

构造函数：创建派生类对象时，必须调用派生类的构造函数，派生类构造函数调用基类的构造函数，以创建派生对象的基类部分。C++11 新增了继承构造函数的机制（使用 using），但默认不继承。

析构函数：释放对象时，先调用派生类析构函数，再调用基类析构函数。

赋值运算符：
编译器不会继承基类的赋值运算符（参数为基类）。但会自动合成隐式定义的赋值运算符（参数为派生类），其功能为调用基类的赋值运算符。

友元函数：不是类成员，故而无法继承

例子：

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

class Base{
public:
    int k = 0;
    void f(){cout << "Base::f()" << endl;}
    Base & operator= (const Base &right){
        if(this != &right){
            k = right.k;
            cout << "operator= (const Base &right)" << endl;
        }
        return *this;
    }
};
class Derive: public Base{};
int main(){
    Derive d, d2;
    cout << d.k << endl; //Base 数据成员被继承

    d.f(); //Base::f()被继承


    Base e;
    //d = e; //编译错误，Base 的赋值运算符不被继承
```

```
    d = d2;  //调用隐式定义的赋值运算符

    return 0;
}
Output：
0
Base::f()
operator= (const Base &right)
```

注意前文提及派生类会自动生成赋值运算符，参数为派生类。也就是仅仅在派生类对象间赋值，基类和派生类相互赋值在不定义类型转换的情况下是不允许的。

2.4 构造与析构

继承与组合在构造以及析构上的执行顺序类似。

基类中的数据成员，通过继承成为派生类对象的一部分，需要在构造派生类对象的过程中调用基类构造函数来正确初始化。

若没有显式调用，则编译器会自动调用基类的默认构造函数。

若想要显式调用，则只能在派生类构造函数的初始化成员列表中进行，既可以调用基类中不带参数的默认构造函数，也可以调用合适的带参数的其他构造函数。先执行基类的构造函数来初始化继承来的数据，再执行派生类的构造函数。

对象析构时，先执行派生类析构函数，再执行由编译器自动调用的基类的析构函数。

例一、隐式调用默认构造函数

```
class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }

        /// 默认构造函数

    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive() { cout << "Derive::Derive()" << endl; }

    /// 无显式调用基类构造函数，则调用基类默认构造函数

};
int main() {
    Derive obj;
    return 0;
} // g++ 1.cpp –o 1.out -std=c++11
```

```
Output：
Base::Base(0)
Derive::Derive()
```

先完成了基类部分的构造，再完成了派生类部分的构造。

例二、显式调用了基类带参数的构造函数

```
class Base
{
   int data;
public:
   Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }
        /// 默认构造函数
   Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
   Derive(int i) : Base(i) { cout << “Derive::Derive()” << endl; }
   /// 显式调用基类构造函数

};
int main() {
   Derive obj(356);
   return 0;
} // g++ 1.cpp –o 1.out -std=c++11
```

显式调用了基类的带参数的构造函数只能在初始化列表中进行。

例四、启用 using 关键字

```
class Base
{
   int data;
public:
   Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
};
class Derive : public Base {
public:
   using Base::Base;      ///相当于 Derive(int i):Base(i){};
};
int main() {
   Derive obj(356);

   return 0;
} // g++ 1.cpp –o 1.out -std=c++11
Output:
```

```
Base::Base(356)
```

在派生类中使用 **using Base::Base**; 来继承基类构造函数，相当于给派生类“定义”了相应参数的构造函数。如果基类里有多个不同参数的构造函数，using 语句会分别构造对应的不同参数的构造函数。

例五、using 一并启用

```
class Base
{
   int data;
public:
   Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
   Base(int i, int j)
      { cout << "Base::Base(" << i << "," << j << ")\n";}
};
class Derive : public Base {
public:
   using Base::Base;      ///相当于 Derive(int i):Base(i){};
                          ///加上 Derive(int i, int j):Base(i,  j){};
};
int main() {
   Derive obj1(356);
   Derive obj2(356, 789);
   return 0;
} // g++ 1.cpp –o 1.out -std=c++11
Output：
Base::Base(356)
Base::Base(356,789)
```

当基类存在多个构造函数时，使用 using 会给派生类自动构造多个相应的构造函数。注意这里是指一个基类有多个构造函数，而不是多重继承。

如果基类的某个构造函数被声明为私有成员函数，则不能在派生类中声明继承该构造函数。（在 L4·创建与销毁·一的 1.2.3 节有叙述过结合委派构造函数将构造函数设置为私有成员函数的例子）

如果派生类使用了继承构造函数，编译器就不会再为派生类生成隐式定义的默认构造函数。

2.5 两类继承方式的选择

2.5.1 public 继承

基类中公有成员仍能在派生类中保持公有。原接口可沿用，最常用。

is-a：基类对象能使用的地方，派生类对象也能使用。

2.5.2 private 继承

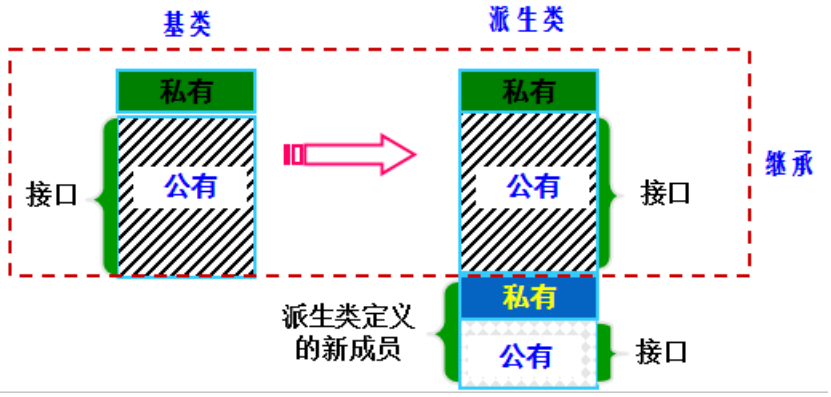
is-implementing-in-terms-of(照此实现)：用基类接口实现派生类功能。移除了 is-a 关系。

通常不使用，用 private 组合替代。可用于隐藏/公开基类的部分接口。公开方法：using 关键字。

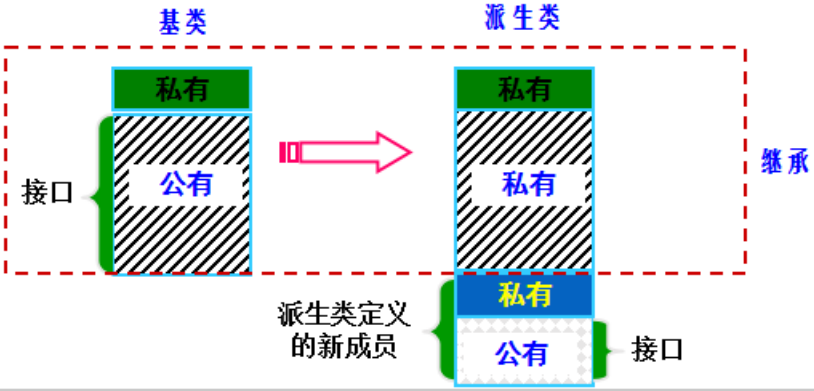
2.6 成员访问权限

基类中的私有成员，不允许在派生类成员函数中访问，也不允许派生类的对象访问它们。（组合当中也是如此，这也一定程度上体现了 private 继承往往可由 private 组合代替）

那么如何访问基类中的私有成员？



情况一，public 继承。可以在类外直接访问基类的公开接口，从而访问积累的私有成员。



情况二、private 继承。和 private 组合同理，在类外既无法直接访问基类的 private 成员，也无法访问基类的 public 成员。（因为继承之后被私有化了）基类的 private 成员无法被派生类的共有接口访问，但是可以被基类的共有部分访问。基类的共有部分不能在类外访问，但是可以被派生类的共有接口访问。

故而我们先设置基类的公有接口访问基类的私有成员，接着设计派生类的公有接口访问基类的公有接口，从而间接实现了访问基类的私有成员。

只要能够理解 private 继承访问基类私有部分需要两次间接访问即可。

基类中的公有成员

允许在派生类成员函数中被访问

若是使用 public 继承方式，则成为派生类公有成员，可以被派生类的对象访问；

若是使用 private/protected 继承方式，则成为派生类私有/保护成员，不能被派生类的对象访问。若想让某成员能被派生类的对象访问，可在派生类 public 部分用关键字 using 声明它的名字。（.h 文件里）或者按照上文计派生类的共有接口访问基类的 public 成员，但是无法直接通过派生类访问基类的 public 成员。

基类中的保护成员

保护成员允许在派生类成员函数中被访问，但不能被外部函数访问。

例一：public 直接继承基类共有接口

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

class Base {
public:
  void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive1: public Base {}; // D1 类的继承方式是 public 继承

int main() {
  Derive1 obj1;
  cout << "calling obj1.baseFunc()..." << endl;

  obj1.baseFunc(); // 基类接口成为派生类接口的一部分，派生类对象可调用

  return 0;
}
Output：
calling obj1.baseFunc()...
in Base::baseFunc()...
```

例二：private 间接访问基类共有接口

```
#include <iostream>
using namespace std;
class Base {
public:
  void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};
class Derive2: private Base

{/// 私有继承，  is-implementing-in-terms-of：用基类接口实现派生类功能

public:
```

```
  void deriveFunc() {
    cout << “in Derive2::deriveFunc(),
                calling Base::baseFunc()..." << endl;
    baseFunc(); /// 私有继承时，基类接口在派生类成员函数中可以使用
  }
};
int main() {
  Derive2 obj2;
  cout << "calling obj2.deriveFunc()..." << endl;
  obj2.deriveFunc();
  //obj2.baseFunc(); ERROR: 基类接口不允许从派生类对象调用
  return 0;
}
```

例三：private 继承结合 using 启用基类共有接口

```
#include <iostream>
using namespace std;
class Base {
public:
  void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive3: private Base {// B 的私有继承
public:
  /// 私有继承时，在派生类 public 部分声明基类成员名字
  using Base::baseFunc;
};
```

```
int main() {
  Derive3 obj3;
  cout << "calling obj3.baseFunc()..." << endl;
  obj3.baseFunc(); //基类接口在派生类 public 部分声明，则派生类对象可调用
  return 0;
}
```

例四、保护成员的访问

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

```
class Base{
private:
  int a{0};
protected:
  int b{0};
};
class Derive : private Base{
public:
  void getA(){cout<<a<<endl;} ///编译错误，不可访问基类中私有成员

  void getB(){cout<<b<<endl;} ///可以访问基类中保护成员

};
int main()
{
  Derive d;
  d.getB();
  //cout<<d.b; ///编译错误，派生类对象不可访问基类中保护成员
  return 0;
}
```

总结：

public 继承

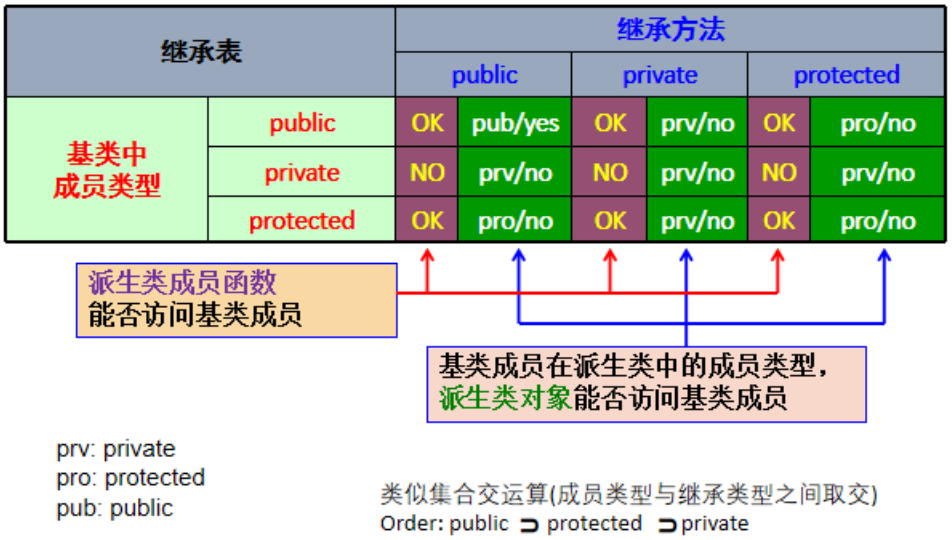
基类的公有成员，保护成员，私有成员作为派生类的成员时，都保持原有的状态。

private 继承

基类的公有成员，保护成员，私有成员作为派生类的成员时，都作为私有成员。

protected 继承

基类的公有成员，保护成员作为派生类的成员时，都成为保护成员，基类的私有成员仍然是私有的。



例题

（不定项选择题）为避免编译错误，下述代码中(1)处可以填写

```
#include <iostream>
using namespace std;
class A {
public:
    int a=1;
protected:
    int b=2;
private:
    int c=3;
};
class B: public A {
public:
    void print() {
        cout << b << endl;
    }
};
int main() {
    B obj_b;
    obj_b.print();
    cout << (1) << endl;
    return 0;
}
```

A obj_b.a　B obj_b.b　C obj_b.c

Protect 对象不可类外访问，但是可以被继承类的成员函数访问。

2.7 组合与继承的联系

优点：支持增量开发，引入新代码而不影响已有代码正确性。

相似点：实现代码重用。，将子对象引入新类，使用构造函数的初始化成员列表初始化。

不同点：

组合——嵌入一个对象以实现新类的功能，has-a 关系，无重写隐藏。

继承——沿用已存在的类提供的接口，is-a 与 is-implementing-in-terms-of，有重写隐藏。

三、重载与重写隐藏

3.1 定义

重载(overload)：
目的：提供同名函数的不同实现，属于静态多态。
函数名必须相同，函数参数必须不同，作用域相同（如位于同一个类中；或同名全局函数）。

重写隐藏(redefining)：
目的：在派生类中重新定义基类函数，实现派生类的特殊功能。
屏蔽了基类的所有其它同名函数。
函数名必须相同，函数参数可以不同。

组合不会发生重写隐藏，但是继承会。

```
#include <iostream>
using namespace std;
class Wheel{
public:
  void inflate(){
    cout<<"Wheel::inflate"<<endl;
  }
  void start(){
    cout<<"Wheel::start"<<endl;
  }
};

class Engine{
public:
  void start(){
    cout<<"Engine::start"<<endl;
  }
  void stop(){}
};
```

```
class Car{
public:
  Engine engine;
  Wheel wheel[4];
   void start(){
    cout<<"Car::start"<<endl;
```

```
  }
};
int main()
{
  Car car;
  car.wheel[0].inflate();
  car.engine.start();
  car.wheel[2].start();
  car.start();
  return 0;
  return 0;
}
Output：
Wheel::inflate
Engine::start
Wheel::start
Car::start
```

在组合中，我们实现了对于每一类的同名函数调用。

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

class Pet{
public:
  void eat(){cout<<"Pet eat"<<endl;}
  void sleep(){cout<<"Pet sleep"<<endl; }
};
class Duck : public Pet{
public:
  void eat(){cout<<"Duck eat"<<endl;}
};
int main()
{
  Duck duck;
  duck.eat();
  duck.sleep();
  return 0;
}
Output：
Duck eat
Pet sleep
```

然而在继承中，很明显 Pet 类的 sleep 函数被派生类重定义，无法被调用。

```
#include <iostream>
```

```
using namespace std;
class T {};
class Base {
public:
  void f() { cout << "B::f()\n"; }
  void f(int i) { cout << "Base::f(" << i << ")\n"; } /// 重载
  void f(double d) { cout << "Base::f(" << d << ")\n"; } ///重载
  void f(T) { cout << "Base::f(T)\n"; } ///重载
};
class Derive : public Base {
public:
  void f(int i) { cout << "Derive::f(" << i << ")\n"; } ///重写隐藏
};
int main() {
  Derive d;
  d.f(10);
  d.f(4.9);    /// 编译警告。执行自动类型转换。
  //  d.f();       /// 被屏蔽，编译错误
  //  d.f(T()); /// 被屏蔽，编译错误
  return 0;
}
```

在 retexter_gcc 编译器上无法警告，但是使用 clang 编译器的警告如下：

```
1869255217/source.cpp:18:7: warning: implicit conversion from 'double' to 'int' changes value from 4.9 to 4 [-Wliteral-conversion]
  d.f(4.9);
    ~ ^~~
1 warning generated.
```

3.2 using 一并启用

和对构造函数的继承一样，using 可以一并启用所有被重写覆盖的同名函数。

```
#include <iostream>
using namespace std;
class T {};
class Base {
```

```
public:
  void f() { cout << "Base::f()\n"; }
  void f(int i) { cout << "Base::f(" << i << ")\n"; }
  void f(double d) { cout << "Base::f(" << d << ")\n"; }
  void f(T) { cout << "Base::f(T)\n"; }
};
class Derive : public Base {
public:
  using Base::f;
  void f(int i) { cout << "Derive::f(" << i << ")\n"; }
};
int main() {
  Derive d;
  d.f(10);
  d.f(4.9);
  d.f();
  d.f(T());
  return 0;
}
output：
Derive::f(10)
Base::f(4.9)
Base::f()
Base::f(T)
```

并且注意到，Derive 和 Base 都有 void f(int i)函数，此处没有发生重定义，而是调用了派生类的 void f(int i)。

另外一种在派生类中调用基类同名函数的方法是直接使用对应的命名空间。

```
#include <iostream>
using namespace std;
class Base {
public:
  void print() { cout << "Base::print()" << endl; }
};
class Derive : public Base {
public:
  void print() { cout << "Derive::print()" << endl; }
};
void fun(Base obj) { obj.print(); }
int main()
{
  Derive d;
  d.Base::print();
```

```
 fun(d);
 return 0;
}
Output：
Base::print()
Base::print()
```

此处调用 d.Base::print();没有导致任何一个同名函数被覆盖。

3.3 总结

所谓的重写隐藏（redefining）其实本质上是一种调用优先级问题。派生类默认优先调用自身的函数。如果不启用 using 关键字，那么除非使用命名空间，否则无法调用基类的函数。哪怕发生类型转换提示 warning 也要调用自身的函数。而启用了 using 关键字后，不使用命名空间时，仍然优先调用自身的函数。但是不同于没有启用 using 关键字的时候，此时如果无法直接调用自身的函数（比如需要类型转换），那么哪怕强制使用派生类的命名空间也会调用基类的函数。

此处启用 using 关键字

```
#include <iostream>
using namespace std;
class T {};
class Base {
public:
 void f() { cout << "Base::f()\n"; }
 void f(int i) { cout << "Base::f(" << i << ")\n"; }
 void f(double d) { cout << "Base::f(" << d << ")\n"; }
 void f(T) { cout << "Base::f(T)\n"; }
};
class Derive : public Base {
public:
 using Base::f;
 void f(int i) { cout << "Derive::f(" << i << ")\n"; }
};
int main() {
 Derive d;
 d.f(10);
 d.Base::f(10);
 d.f(4.9);
 d.Derive::f(4.9);
 d.f();
 d.f(T());
 return 0;
}
output：
```

```
Derive::f(10)
Base::f(10)
Base::f(4.9)
Base::f(4.9)
Base::f()
Base::f(T)
```

此处不启用 using 关键字

```
#include <iostream>
using namespace std;
class T {};
class Base {
public:
  void f() { cout << "Base::f()\n"; }
  void f(int i) { cout << "Base::f(" << i << ")\n"; }
  void f(double d) { cout << "Base::f(" << d << ")\n"; }
  void f(T) { cout << "Base::f(T)\n"; }
};
class Derive : public Base {
public:
  void f(int i) { cout << "Derive::f(" << i << ")\n"; }
};
int main() {
  Derive d;
  d.f(10);
  d.Base::f(10);
  d.f(4.9);
  d.Base::f(4.9);
  d.Base::f();
  d.Base::f(T());
  return 0;
}
Output：
Derive::f(10)
Base::f(10)
Derive::f(4)
Base::f(4.9)
Base::f()
Base::f(T)
```

不启用 using 关键字时直接调用 f()和 f(T)是不合法的。

备注：此处对类内命名空间的表达有误，有待进一步更正。

3.4 using 关键字作用总结

3.4.1 （在派生类中使用）继承基类构造函数、恢复被屏蔽的基类成员函数（每次只恢复一个函数，而不是基类的所有函数）

3.4.2 指示命名空间 using namespace std;

3.4.3 将另一个命名空间的成员引入当前命名空间
如：using std::cout; cout << endl;

3.4.4 定义类型别名，如：using a = int;

例题

关于下列代码的说法正确的是（\n为换行符）

```
#include <iostream>
using namespace std;
class A {
public:
  int data;
  A(int d)
    {cout << "A::A(" << d << ")\n";}
  void f(double d)
    {cout << "A::f(" << d << ")\n";}
protected:
  void g(){}
};
```

```
class B: public A {
public:
  int data{2017};
  using A::A;
  void f(){}
  void print(){
    cout << "data = " << data << endl;
  }
};
int main() {
  B b(6);
  b.print();
  return 0;
}
```

A main函数中可通过b.g();语句调用函数A::g()

B 去掉B类定义中的using A::A;语句，程序会出现编译错误

C 程序的运行结果为A::A(6)\ndata=2017\n

D main函数中可通过b.f(17.315);调用函数A::f(double d)

提交

47

D 选项，注意到 B 中只 using 了 A 的构造函数，没有恢复 A 的 f 函数。

四、多重继承

4.1 定义、意义与潜在风险

派生类同时继承多个基类

格式：

```
class Derive : public MiddleA, public MiddleB{
};
```

数据存储风险

如果派生类 D 继承的两个基类 A,B，是同一基类 Base 的不同继承，则 A,B 中继承自 Base 的数据成员会在 D 有两份独立的副本，可能带来数据冗余。

二义性风险

如果派生类 D 继承的两个基类 A,B，有同名成员函数或同名成员数据 a，则访问 D 中 a 时，编译器无法判断要访问的哪一个基类成员。

二义性例子

```
#include <iostream>
using namespace std;
class Base {
public:
  int a{0};
};
class MiddleA : public Base {
public:
  void addA() { cout << "a=" << ++a << endl; };
  void bar() { cout << "A::bar" << endl; };
};
class MiddleB : public Base {
public:
  void addB() { cout << "a=" << ++a << endl; };
  void bar() { cout << "B::bar" << endl; };
};

class Derive : public MiddleA, public MiddleB{
};
int main(){
    Derive d;
    d.bar();
}
output :
main.cpp:22:7: error: member 'bar' found in multiple base classes of different types
       d.bar();
         ^
main.cpp:10:8: note: member found by ambiguous name lookup
    void bar() { cout << "A::bar" << endl; };
           ^
main.cpp:15:8: note: member found by ambiguous name lookup
    void bar() { cout << "B::bar" << endl; };
           ^
1 error generated.
```

稍作修改

```
#include <iostream>
using namespace std;
class Base {
public:
```

```
  int a{0};
};
class MiddleA : public Base {
public:
  void addA() { cout << "a=" << ++a << endl; };
  void abar() { cout << "A::bar" << endl; };
};
class MiddleB : public Base {
public:
  void addB() { cout << "a=" << ++a << endl; };
  void bbar() { cout << "B::bar" << endl; };
};

class Derive : public MiddleA, public MiddleB{
};
int main(){
    Derive d;
    d.abar();
    d.bbar();
}
output :
A::bar
B::bar
```

例二、同名成员操作

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

class Base {
public:
  int a{0};
};
class MiddleA : public Base {
public:
  void addA() { cout << "a=" << ++a << endl; };
  void bar() { cout << "A::bar" << endl; };
};
class MiddleB : public Base {
public:
  void addB() { cout << "a=" << ++a << endl; };
  void bar() { cout << "B::bar" << endl; };
};
class Derive : public MiddleA, public MiddleB{
};
```

```
int main() {
  Derive d;
  d.addA(); /// 输出 a=1。

  d.addB(); /// 仍然输出 a=1。

  d.addB(); /// 输出 a=2。
  cout << d.MiddleA::a << endl;
/// 输出 A 中的成员 a 的值 1
 cout << d.MiddleB::a << endl;
/// 输出 B 中的成员 a 的值 2
  d.MiddleA::bar();
  d.MiddleB::bar();
  return 0;
}
output :
a=1
a=1
a=2
1
2
A::bar
B::bar
```

注意到，多重继承的二义性并没有导致重复定义，编译器通过命名空间区分了二者不同的 a 与 bar。但是不加命名空间，则无法区分。