第4章-类之间的关系

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

类之间的关系大致可分为两种，横向关系 和 纵向关系

1. 类之间的横向关系

1.1组合（复合）

它是一种 "is a part of" 的关系， 部分与整体，包含与被包含。组合是一个类中包含另一个类对象。相比聚合，组合是一种强所属关系，组合关系的两个对象往往具有相同的生命周期，被组合的对象是在组合对象创建的同时或者创建之后创建，在组合对象销毁之前销毁。一般来说被组合对象不能脱离组合对象独立存在，整体不存在，部分一定不存在。

举例：人与手、人与头之间的关系，人需要包含头和手，头、手是人的一部分且不能脱离人独立而存在。

在C++语法中，通常在组合类中包含被组合类对象来实现组合关系：

```
class CHand{};
class CPeople{
    CHand m_hand;  //组合（复合）关系
};
```

1.2 依赖

它是一种 "uses a" 的关系。一个对象的某种行为依赖于另一个类对象，被依赖的对象视为完成某个功能的工具，并不持有对他的引用，只有在完成某个功能的时候才会用到，而且是必不可少的。依赖之间是没有生命周期约束关系的。

举例：人要完成编程这件事，那么需要用到电脑，电脑作为一个工具，其他的时候不需要，电脑也不可能作为人的属性而存在（非组合关系），人必须依赖于电脑才能完成编程这件事。

C++语法中，代码的表现形式为多种，通常将被依赖的对象作为另一类方法的参数的形式实现两个类之间的依赖关系，。

```
class CComputer{};
class CPeople{
   void Code(CComputer *pc)  //或：CComputer &pc，
   {}
};
```

1.3 关联

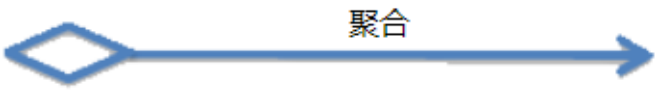
它是一种"has a"的关系。关联不是从属关系，而是平等关系，可以拥有对方，但不可占有对方。完成某个功能与被关联的对象有关，但是可有可无。被关联的对象与关联的对象无生命周期约束关系，被关联对象的生命周期由谁创建就由谁来维护。只要二者同意，可以随时解除关系或是进行关联，被关联的对象还可以再被别的对象关联，所以关联是可以共享的。

举例：人和朋友的关系，人要完成玩游戏这个功能，没有朋友可以自己玩游戏，如果交到朋友了就可以和朋友一起玩游戏。
C++语法中，通常在关联的类中定义被关联类对象的指针形式实现两个类之间的关联关系。

```
class CFriend{};
class CPeople{
   CFriend *m_pFriend;  //关联关系
};
```

1.4 聚合

它是一种"owns a"的关系。多个被聚合的对象聚集起来形成一个大的整体，聚合的目的是为了统一进行管理同类型的对象，聚合是一种弱所属关系，被聚合的对象还可以再被别的对象关联，所以被聚合对象是可以共享的。虽然是共享的，聚合代表的是一种更亲密的关系，相当于强版本的关联。



举例：一堆人组成一个家庭，进行统一管理完成敲代码工作。
C++语法中，通常在聚合类中定义被聚合对象指针的数组、链表等容器。

```
class CPeople{};
class CFamily{
```

```
    CPeople* m_pFamily[10];
};
```

2. 类之间的纵向关系

2.1 继承

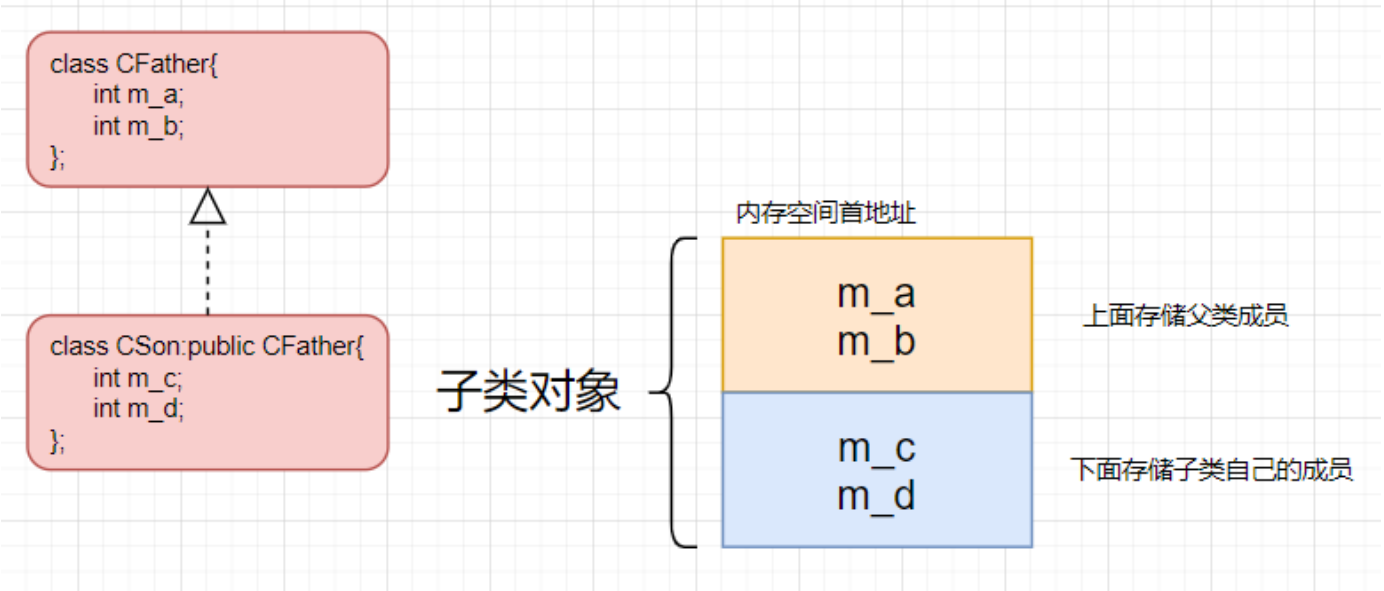
被继承的类叫做基类（父类），继承的类叫派生类（子类），在派生类类名后面加：继承方式 基类

```
class CFather{};
class CSon:public CFather{};
```

通过继承关系，子类可以使用父类的成员。如果子类和父类有同名的成员，默认使用子类的成员，如果想要使用父类的成员，需要在成员名前加上类名::用于显式的指定区分，

```
son.m_a;            //子类成员
son.CSon::m_a;      //子类成员
son.CFather::m_a;   //父类成员
```

子类继承父类，相当于将父类的成员包含到自己的类里，所以定义子类对象所占用的空间大小除了子类自身的成员还包括父类的成员。成员在内存空间分布为：先父类成员后子类成员，而每个类中的成员分布与在类中声明的顺序一致。



2.2 继承下构造析构执行的顺序

定义子类对象时执行顺序：**父构造->子构造->孙构造...| ...孙析构->子析构->父析构。**

构造顺序说明：在子类创建对象的时候，调用子类的构造函数（注意这里并不是直接先执行父类的构造函数），但要先执行构造的初始化列表，在初始化列表中会默认调用父类的无参构造初始化父类成

员，如父类只有带参数的构造，在子类的初始化参数列表中必须显式的指定父类构造进行初始化。这有点像之前说的组合关系形式。

析构顺序说明：子类对象的生命周期结束后，因为是子类所以自动调用子类析构，当析构执行完了，才会回收对象分配的空间，当然这个空间包含创建的父类的成员，那么回收父类成员前，自动调用父类的析构。如果是new出来的子类对象，同理。

继承的优点：我们可以将类中的一些功能相近、相似的共同的方法，抽离出来放到单独的一个类中，并让其继承这个类，那么抽离出来的类就是父类，将来其他类在增加公共的方法时，我只需要在父类添加一份即可。提高了代码的复用性、扩展性。

2.3 继承方式

三种继承方式 public、protected、private：
继承方式描述了父类成员在子类中的访问控制，即所能使用的一个范围。继承方式与访问修饰符共同决定了父类成员在子类中所表现的属性。

继承方式	父类中的属性	子类中的属性
public	public	public
	protected	protected
	private	不可访问
protected	public	protected
	protected	protected
	private	不可访问
private	public	private
	protected	private
	private	不可访问

2.4 父类的指针指向子类的对象

对于函数重载而言，我们调用的时候，可以根据参数类型、参数个数，编译器自动匹配调用哪个函数。
同样如果在一个类中存在两个同名函数（参数列表不同），那么也可以根据调用者传递的参数自动的区分执行哪个函数，因为也是一个函数重载的关系。

```
class CTest {
    //函数重载，可根据参数自动匹配
    void fun();

```

```
5       void fun(int a);
    };
```

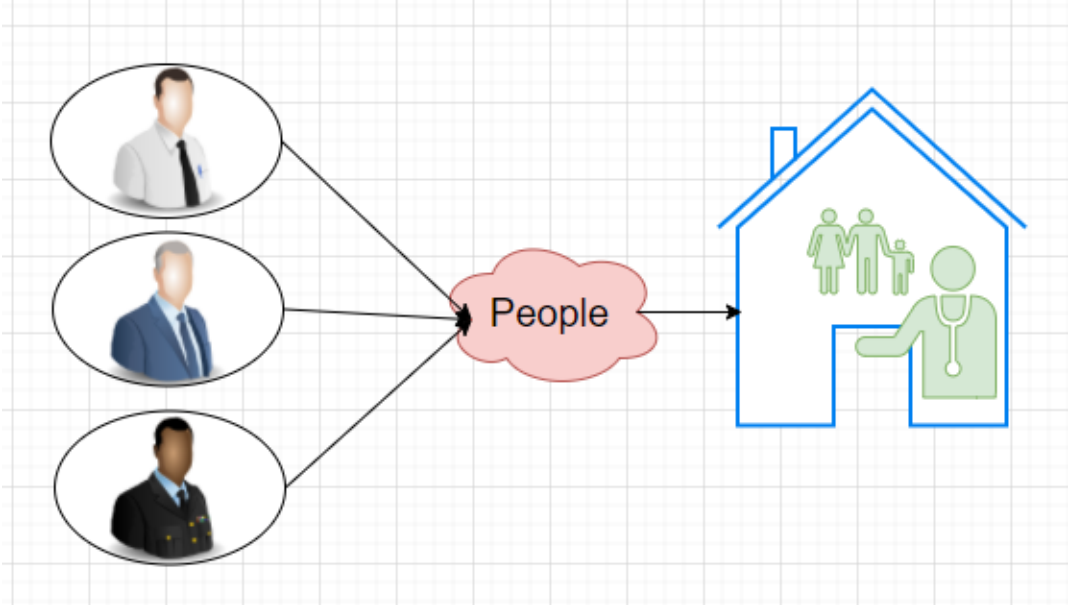
那对于父类和子类中，如果有同名的函数但是参数列表不同，则不能够自动区分匹配，因为他们之间的关系并不是函数重载的关系，作用域不同，必须使用 类名:: 去区分到底该调用哪个函数。这种关系一般称之为 **隐藏。**

```
1   class CFather {
2       void fun();
3   };
4   class CSon :public CFather {
5       void fun(int a);    //隐藏关系，自动将父类的成员屏蔽了
6   };
```

在继承关系下，允许父类的指针指向子类的对象，但是反过来却不行。

```
1   CFather * p = new CSon;
```

这么做的好处是：父类的指针可以统一多个类的类型，提高代码的复用性、扩展性。



2.5 类成员函数指针

调用函数的两种方式：通过函数名直接调用、通过函数指针间接调用。
通过函数指针调用的好处：真正的函数指针可以将实现同一功能的多个模块统一起来标识，使系统结构更加清晰，后期更容易维护。或者归纳为：便于分层设计、利于系统抽象、降低耦合度以及使接口与实现分开，提高代码的复用性、扩展性。

定义函数指针变量

```
void(*p_fun)(int)=&fun;  //定义变量并初始化
p_fun = &fun;            //赋值，指向一个函数
(*p_fun)(10);           //间接调用函数
```

定义函数指针有时看起来比较繁琐、可读性差一些，我们可以用typedef进行优化。

```
typedef void(*P_FUN)(int);
P_FUN p_fun = &fun;
```

类成员函数与普通的函数的区别：

1. 所属的作用域不同，类成员函数标识了所属的类，必须通过对象调用（虽然可以是空指针对象，但必须得有）。
2. 类成员函数编译器会默认加上一个隐藏的参数: this指针。
   所以定义类成员函数的指针与普通的函数指针肯定会有所区别：
   C++ 提供了三种运算符 `::*` 、`.*`、 `->*` 用于定义和使用类成员函数指针。

```
void (CTest::*p_fun)() = &CTest::show;  //定义类成员函数指针并初始化，注意： & 和 类名作用域 都不能省略，

typedef void (CTest::*P_FUN)();   //使用typedef 进行优化
P_FUN p_fun2 = &CTest::show;

CTest tst;
CTest*pTst = new CTest;

(tst.*p_fun2)();                  //普通对象通过指针调用类成员函数
(pTst->*p_fun2)();                //指针对象通过指针调用类成员函数
```

可以用类成员函数指针来模拟实现多态。

```
class CPeople {};

class CYellow:public CPeople {
public:
    void Eat() {
        cout << "元气猫" << endl;
    }
};
int main(){
    CPeople* pPeo = new CYellow;
    //pPeo->Eat();   //不能直接调用
```

```
12
13      void (CPeople::* p_fun)() = (void (CPeople:: * )())&CYellow::Eat;
    //通过强转指向子类的函数
14      (pPeo->*p_fun)();   //元气猫
15      return 0;
16  }
```