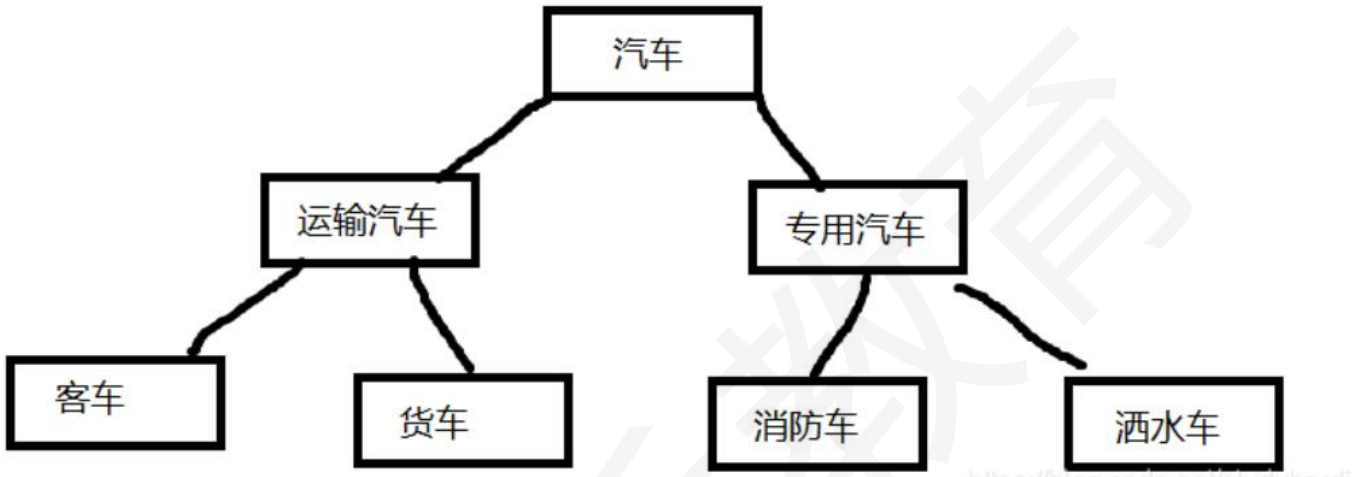
第四章 继承

4.1 为什么需要继承?

继承性是一个非常自然的概念，现实世界中的许多事物都是具有继承性的。人们一般用层次分类的方法来描述 他们的关系。例如，下图就是一个简单的汽车分类图：



在这个分类树中建立了一个层次结构，最高一层是最普遍、最一般的，每一层都比它的前一层更具体，低层具 高层的特性，同时也有与高层的细微不同。例如，确定了一辆车是客车以后，没有必要再指出它可以进行运输，因为客车本身就是从运输汽车类中派生出来的。

4.2 继承的概念

1、继承机制是面向对象程序设计使代码可以复用的最重要的手段，它允许程序员在保持原有的特性基础上进行扩展，增加功能。

2、一个B类继承于A类，或称从类A派生类B。这样的话，类A成为基类（父类）， 类B成为派生类（子类）。

派生类中的成员，包含两大部分：

- 一类是从基类继承过来的，一类是自己增加的成员。
- 从基类继承过过来的表现其共性，而新增的成员体现了其个性。

4.3 派生类定义

```
Class 派生类名 ：  继承方式 基类名
{
     //派生类新增的数据成员和成员函数
```

```
}
```

三种继承方式：

- public ： 公有继承
- private ： 私有继承
- protected ： 保护继承

```
class Person
{
public:
        void Print(){
                cout<<"name:"<<_name<<endl;
                cout<<"age:"<<_age<<endl;
        }
protected:
        string _name; //姓名
        int _age; //年龄
};
/*继承后父类的Person的成员（成员函数+变量）都会变成子类的一部分。这里
体现出了Student和Teacher复用了Person的成员。*/
class Student: public Person
{
protected:
        int _stuid; //学号
};

class Teacher:public Person
{
protected:
        int _jobid; //工号
};

int main()
{
    return 0;
}
```

派生类/子类继承了基类/父类中的所有的成员变量和函数，验证：

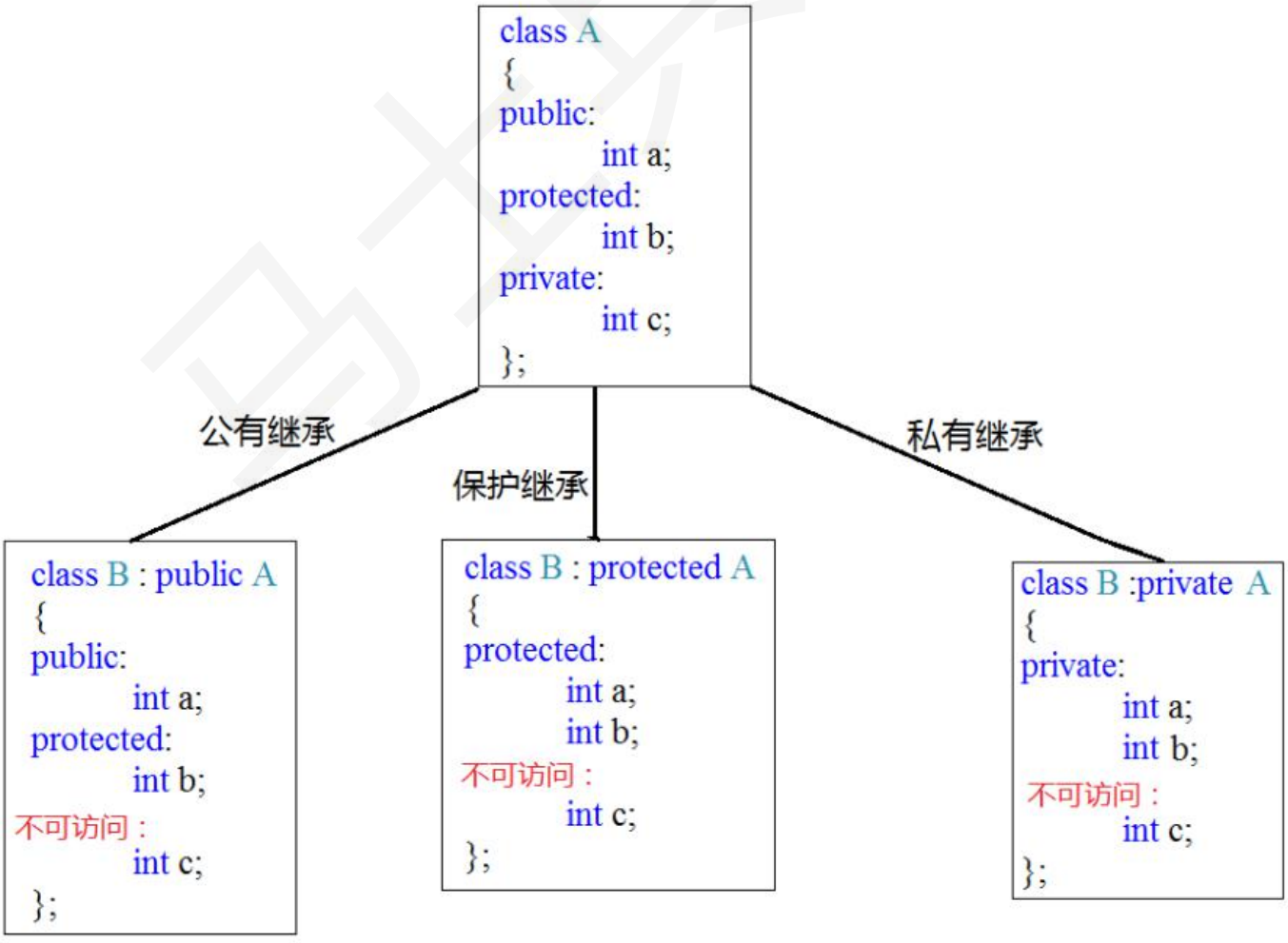
```
Student s;
Teacher t;
cout << sizeof(s) << endl;
cout << sizeof(t) << endl;

s.Print();
t.Print();
```

4.4 派生类访问控制

1、C++中的继承方式会影响子类的对外访问属性

> public继承：父类成员在子类中保持原有访问级别
>
> private继承：父类成员在子类中变为private成员
>
> protected继承：父类中public成员会变成protected，父类中protected成员仍然为protected，父类中private成员仍然为private

	父类成员访问级别			
继承方式		public	protected	private
	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private



2、private成员在子类中依然存在，但是却无法访问到。不论何种方式继承基类，派生类都不能直接使用基类的私有成员 。

```
#include <iostream>

using namespace std;

class A
{
        public:
                int x;
        protected:
                int y;
        private:
                int z;
    public:
        void func_a(){cout << "func_a" << endl;}
};

//派生类B继承在A
class B: public A
{
/*
        public:
                int x;
        protected:
                int y;
        private:
                int z;
                */
        public:
                void handle_attrib()
                {
                        this->y = 100;
                //      this->z = 1000;
                }
};

class C: private A
{
/*
        private:
                int x;
        private:
                int y;
        private:
                int z;
                */
        public:
                void handle_attrib()
                {
```

```
        this->x = 100;
        this->y = 100;
```

4.5 单继承中构造函数和析构函数的调用顺序

- 子类对象在创建时会首先调用父类的构造函数
- 父类构造函数执行完毕后，才会调用子类的构造函数
- 当父类构造函数有参数时，需要在子类初始化列表(参数列表)中显示调用父类构造函数
- 析构函数调用顺序和构造函数相反

```
#include <iostream>

using namespace std;

//基类
class A
{
protected:
    int z;
private:
    int x;

public:
    A()
    {cout << "A()" << endl;}
    ~A()
    {cout << "~A()" << endl;}

    A(int a, int b):z(a), x(b)
    {cout << "A(int a, int b):z(a), x(b)" << endl;}

    int get_z() {return z;}
    int get_x() {return x;}
    void set_x(int x)
    {this->x = x;}
};

class B : public A
{
public:
    B()
    {cout << "B()" << endl;}
    ~B()
    {cout << "~B()" << endl;}

    //在派生类中指定调用基类中某个构造函数
    B(int a, int b):A(a, b)
    {
       // z = a;
```

```
        // set_x(b);
         cout << "B(int a, int b)" << endl;
    }
};

int main()
{
    //构造一个派生类的对象
  //  B b;
    B t(10, 20);
```

4.6 派生类中的成员变量和基类中的成员变量名冲突

```
class A
{
        public:
                int x;
        protected:
                int y;
        private:
                int z;
        public:
                A(){cout << "基类的构造函数" << endl;}
                ~A(){cout << "基类的析构函数" << endl;}
};
class E:public A
{
        public:
            int x;
                E(){cout << "派生类的构造函数" << endl;};
                ~E(){cout << "派生类的析构函数" << endl;};
};
int main()
{
        E e;
       cout << "sizeof(E): " << sizeof(E) << endl; //16
       return 0;
 }
```

从以上代码的执行结果中我们可以得出结论：

> 派生类中的成员变量和基类中的同名的成员变量在派生类中是共存的

那么如果使用派生对象访问x变量，最终访问到的是基类中的x还是派生类中的呢？

```
class A
{
        public:
                int x;
```

```
        protected:
                int y;
        private:
                int z;
        public:
                A(){this->x = 1;cout << "基类的构造函数" << endl;}
                A(int x):x(x){cout << "基类的构造函数A(int x)" << endl;}
                ~A(){cout << "基类的析构函数" << endl;}

        void func_a(){cout << "func_a" << endl;}
};

class E:public A
{
        public:
                int x;
                E(){cout << "派生类的构造函数" << endl;};
                E(int a):x(a){cout << "派生类的构造函数E(int a)" << endl;};
                ~E(){cout << "派生类的析构函数" << endl;};
};
int main()
{
        E e(10);
        cout << e.x << endl;
        cout << e.A::x << endl; //  基类名::基类中的成员变量
        cout << "sizeof(E): " << sizeof(E) << endl;
    return 0;
}
```

总结:

- 当子类成员和父类成员同名时，子类依然从父类继承同名成员
- 如果子类有成员和父类同名，子类访问其成员默认访问子类的成员(本作用域，就近原则)
- 在子类通过作用域::进行同名成员区分(在派生类中使用基类的同名成员，显示使用类名限定符)

4.7 隐藏

1 子类函数与父类函数的名称相同，参数也相同，但是父类函数没有virtual，父类函数被隐藏

2 子类的函数与父类的名称相同，但是参数不同，父类函数被隐藏

```
class Father
{
public:
    void func1()
    {cout << "father func1" << endl;}

    void func2(int x)
    {cout << "father func2" << endl;}
```

```
};

class Son : public Father
{
public:
    void func1()
    {cout << "son func1" << endl;}

    void func2(char x)
    {cout << "son func2" << endl;}
};
```

4.8 继承中的静态成员特性

1、静态成员函数和非静态成员函数的共同点

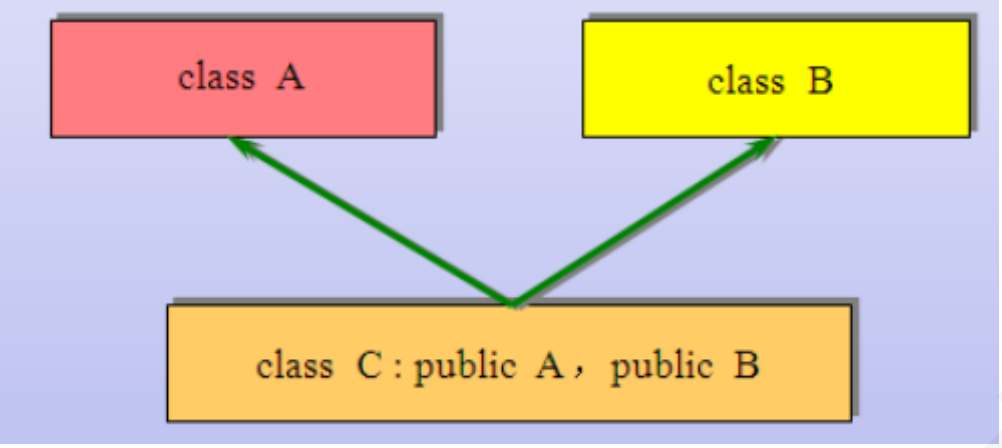
- 他们都可以被继承到派生类中
- 如果重新定义一个静态成员函数，所有在基类中的其他重载函数会被隐藏
- 如果我们改变基类中一个函数的特征，所有使用该函数名的基类版本都会被隐藏

2、静态成员函数不能是虚函数（virtual function）

```
class Base{
public:
        static int getNum(){ return sNum; }
        static int getNum(int param){
                return sNum + param;
        }
public:
        static int sNum;
};
int Base::sNum = 10;

class Derived : public Base{
public:
        static int sNum; //基类静态成员属性将被隐藏
#if 0
        //重定义一个函数，基类中重载的函数被隐藏
        static int getNum(int param1, int param2){
                return sNum + param1 + param2;
        }
#else
        //改变基类函数的某个特征，返回值或者参数个数，将会隐藏基类重载的函数
        static void getNum(int param1, int param2){
                cout <<  sNum + param1 + param2 << endl;
        }
#endif
};
int Derived::sNum = 20;
```

4.9 多继承

1、多继承的概念：我们可以从一个类继承，我们也可以能同时从多个类继承，这就是多继承。



2、多继承下派生类的定义格式如下：

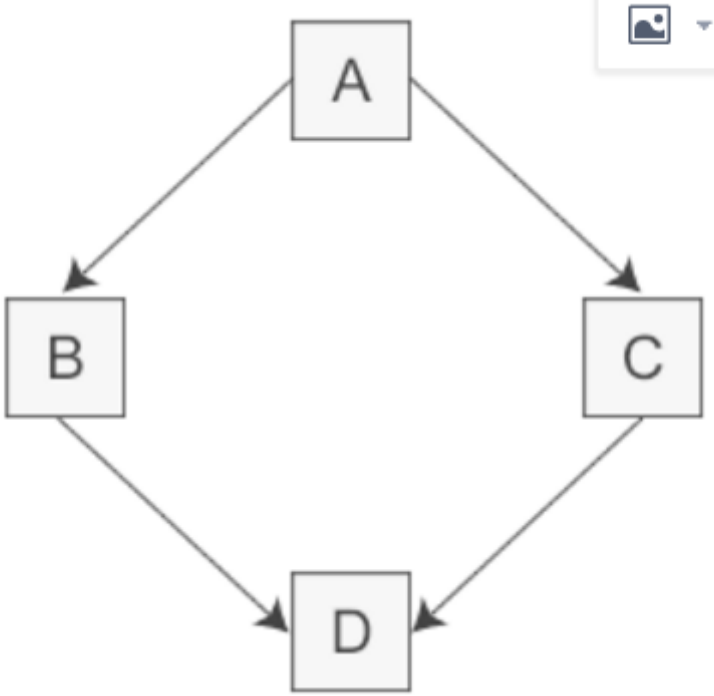
> class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,...
>
> {
>
> <派生类类体>
>
> };

```
class A
{
…
};
class B
{
…
};
class C : public A, public B
{
…
};
```

4.10 菱形继承/环形继承

多继承（Multiple Inheritance）是指从多个直接基类中产生派生类的能力，多继承的派生类继承了所有父类的成员。尽管概念上非常简单，但是多个基类的相互交织可能会带来错综复杂的设计问题，命名冲突就是不可回避的一个。

多继承时很容易产生命名冲突，即使我们很小心地将所有类中的成员变量和成员函数都命名为不同的名字，命名冲突依然有可能发生，比如典型的是菱形继承，如下图所示：



类 A 派生出类 B 和类 C，类 D 继承自类 B 和类 C，这个时候类 A 中的成员变量和成员函数继承到类 D 中变成了两份，一份来自 A-->B-->D 这条路径，另一份来自 A-->C-->D 这条路径。

在一个派生类中保留间接基类的多份同名成员，虽然可以在不同的成员变量中分别存放不同的数据，但大多数情况下这是多余的：因为保留多份成员变量不仅占用较多的存储空间，还容易产生命名冲突。假如类 A 有一个成员变量 a，那么在类 D 中直接访问 a 就会产生歧义，编译器不知道它究竟来自 A -->B-->D 这条路径，还是来自 A-->C-->D 这条路径。下面是菱形继承的具体实现：

```
//间接基类A
class A{
protected:
    int m_a;
};
//直接基类B
class B: public A{
protected:
    int m_b;
};
//直接基类C
class C: public A{
protected:
    int m_c;
};
//派生类D
class D: public B, public C{
public:
    void seta(int a){ m_a = a; }  //命名冲突
    void setb(int b){ m_b = b; }  //正确
    void setc(int c){ m_c = c; }  //正确
    void setd(int d){ m_d = d; }  //正确
private:
```

```
    int m_d;
};
int main(){
    D d;
    return 0;
}
```

这段代码实现了上图所示的菱形继承，第 25 行代码试图直接访问成员变量 m_a，结果发生了错误，因为类 B 和类 C 中都有成员变量 m_a（从 A 类继承而来），编译器不知道选用哪一个，所以产生了歧义。

为了消除歧义，我们可以在 m_a 的前面指明它具体来自哪个类：

> void seta(int a){ B::m_a = a; }

这样表示使用 B 类的 m_a。当然也可以使用 C 类的：

> void seta(int a){ C::m_a = a; }

6.6 虚继承

为了解决多继承时的命名冲突和冗余数据问题，C++ 提出了虚继承，使得在派生类中只保留一份间接基类的成员。

在继承方式前面加上 virtual 关键字就是虚继承，请看下面的例子：

```
//间接基类A
class A{
protected:
    int m_a;
};
//直接基类B
class B: virtual public A{  //虚继承
protected:
    int m_b;
};
//直接基类C
class C: virtual public A{  //虚继承
protected:
    int m_c;
};
//派生类D
class D: public B, public C{
public:
    void seta(int a){ m_a = a; }  //正确
    void setb(int b){ m_b = b; }  //正确
    void setc(int c){ m_c = c; }  //正确
    void setd(int d){ m_d = d; }  //正确
private:
    int m_d;
```

```
};
int main(){
    D d;
    return 0;
}
```

这段代码使用虚继承重新实现了上图所示的菱形继承，这样在派生类 D 中就只保留了一份成员变量 m_a，直接访问就不会再有歧义了。

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。其中，这个被共享的基类就称为虚基类（Virtual Base Class），本例中的 A 就是一个虚基类。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一份虚基类的成员。

现在让我们重新梳理一下本例的继承关系，如下图所示：