Lesson01--C++中的继承

【本节目标】

- 1.继承的概念及定义
- 2.基类和派生类对象赋值转换
- 3.继承中的作用域
- 4.派生类的默认成员函数
- 5.继承与友元
- 6.继承与静态成员
- 7.复杂的菱形继承及菱形虚拟继承
- 8.继承的总结和反思
- 9.笔试面试题

1.继承的概念及定义

1.1继承的概念

> **继承**(inheritance)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**保持原有类特性的基础上进行扩展**，增加功能，这样产生新的类，称派生类。继承**呈现了面向对象程序设计的层次结构**，体现了由简单到复杂的认知过程。以前我们接触的复用都是函数复用，**继承是类设计层次的复用**。

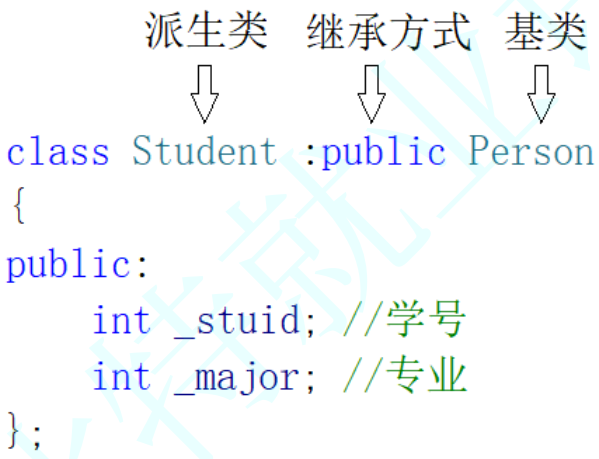
```
class Person
{
public:
    void Print()
    {
        cout << "name:" << _name << endl;
        cout << "age:" << _age << endl;
    }
protected:
    string _name = "peter"; // 姓名
    int _age = 18;  // 年龄
};

// 继承后父类的Person的成员（成员函数+成员变量）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。下面我们使用监视窗口查看Student和Teacher对象，可以看到变量的复用。调用Print可以看到成员函数的复用。
class Student : public Person
{
protected:
    int _stuid; // 学号
};
```

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

int main()
{
    Student s;
    Teacher t;
    s.Print();
    t.Print();

    return 0;
}
```

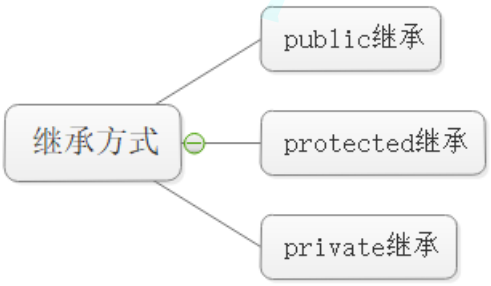
1.2 继承定义

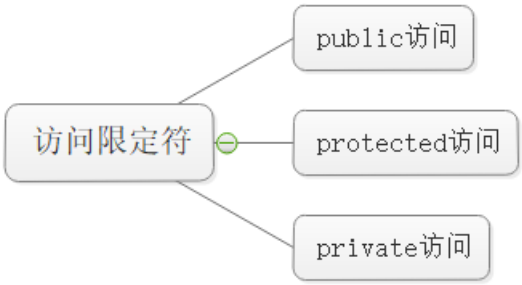
1.2.1定义格式

下面我们看到Person是父类，也称作基类。Student是子类，也称作派生类。



1.2.2继承关系和访问限定符





1.2.3继承基类成员访问方式的变化

类成员/继承方式	public继承	protected继承	private继承
基类的public成员	派生类的public成员	派生类的protected成员	派生类的private成员
基类的protected成员	派生类的protected成员	派生类的protected成员	派生类的private成员
基类的private成员	在派生类中不可见	在派生类中不可见	在派生类中不可见

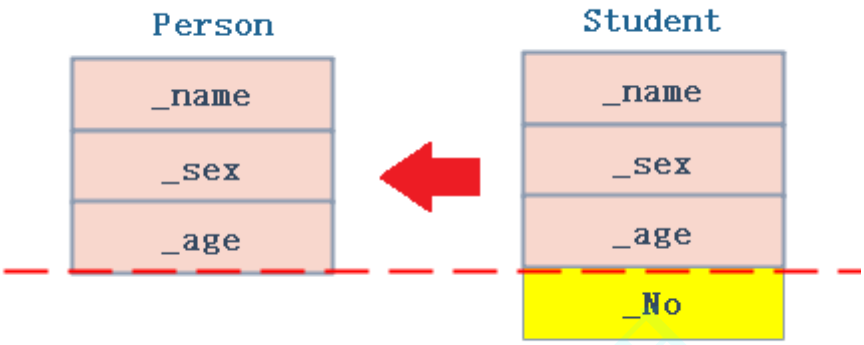
总结:

1. 基类private成员在派生类中无论以什么方式继承都是不可见的。这里的**不可见是指基类的私有成员还是被继承到了派生类对象中，但是语法上限制派生类对象不管在类里面还是类外面都不能去访问它。**
2. 基类private成员在派生类中是不能被访问，如果基类成员不想在类外直接被访问，但需要在派生类中能访问，就定义为protected。**可以看出保护成员限定符是因继承才出现的。**
3. 实际上面的表格我们进行一下总结会发现，基类的私有成员在子类都是不可见。基类的其他成员在子类的访问方式 == Min(成员在基类的访问限定符，继承方式)，public > protected > private。
4. 使用关键字class时默认的继承方式是private，使用struct时默认的继承方式是public，**不过最好显示的写出继承方式。**
5. **在实际运用中一般使用都是public继承，几乎很少使用protetced/private继承**，也不提倡使用protetced/private继承，因为protetced/private继承下来的成员都只能在派生类的类里面使用，实际中扩展维护性不强。

```
// 实例演示三种继承关系下基类成员的各类型成员访问关系的变化
class Person
{
public :
    void Print ()
    {
        cout<<_name <<endl;
    }
protected :
    string _name ; // 姓名
private :
    int _age ;   // 年龄
};

//class Student : protected Person
//class Student : private Person
class Student : public Person
{
protected :
    int _stunum ; // 学号
};
```

2.基类和派生类对象赋值转换

- **派生类对象** 可以赋值给 **基类的对象 / 基类的指针 / 基类的引用。**这里有个形象的说法叫切片或者切割。寓意把派生类中父类那部分切来赋值过去。
- 基类对象不能赋值给派生类对象。
- 基类的指针或者引用可以通过强制类型转换赋值给派生类的指针或者引用。但是必须是基类的指针是指向派生类对象时才是安全的。这里基类如果是多态类型，可以使用RTTI(Run-Time Type Information)的dynamic_cast 来进行识别后进行安全转换。（ps：这个我们后面再讲解，这里先了解一下）




```
class Person
{
protected :
    string _name; // 姓名
    string _sex;  // 性别
    int _age;    // 年龄
};
class Student : public Person
{
public :
    int _No ; // 学号
};
void Test ()
{
    Student sobj ;
    // 1.子类对象可以赋值给父类对象/指针/引用
    Person pobj = sobj ;
    Person* pp = &sobj;
    Person& rp = sobj;

    //2.基类对象不能赋值给派生类对象
    sobj = pobj;

    // 3.基类的指针可以通过强制类型转换赋值给派生类的指针
    pp = &sobj
    Student* ps1 = (Student*)pp; // 这种情况转换时可以的。
    ps1->_No = 10;

    pp = &pobj;
    Student* ps2 = (Student*)pp; // 这种情况转换时虽然可以，但是会存在越界访问的问题
    ps2->_No = 10;
}
```

3.继承中的作用域

1. 在继承体系中**基类**和**派生类**都有**独立的作用域。**
2. 子类和父类中有同名成员，**子类成员将屏蔽父类对同名成员的直接访问，这种情况叫隐藏，也叫重定义。**（在子类成员函数中，可以**使用 基类::基类成员 显示访问**）
3. 需要注意的是如果是成员函数的隐藏，只需要函数名相同就构成隐藏。
4. 注意在实际中在**继承体系里**面最好**不要定义同名的成员。**

```
// Student的_num和Person的_num构成隐藏关系，可以看出这样代码虽然能跑，但是非常容易混淆
class Person
{
protected :
    string _name = "小李子"; // 姓名
    int _num = 111;         // 身份证号
};

class Student : public Person
{
public:
    void Print()
    {
        cout<<" 姓名:"<<_name<< endl;
        cout<<" 身份证号:"<<Person::_num<< endl;
        cout<<" 学号:"<<_num<<endl;
    }
protected:
    int _num = 999; // 学号
};

void Test()
{
    Student s1;
    s1.Print();
};
```

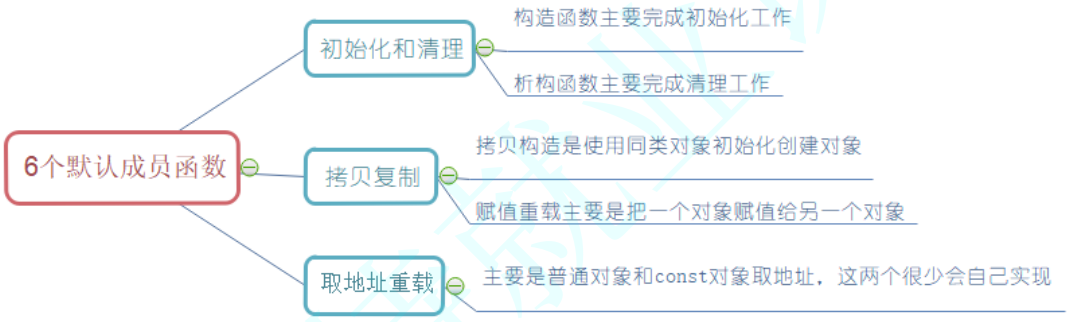
```
// B中的fun和A中的fun不是构成重载，因为不是在同一作用域
// B中的fun和A中的fun构成隐藏，成员函数满足函数名相同就构成隐藏。
class A
{
public:
    void fun()
    {
        cout << "func()" << endl;
    }
};
class B : public A
{
public:
    void fun(int i)
    {
        A::fun();
        cout << "func(int i)->" <<i<<endl;
    }
};

void Test()
```

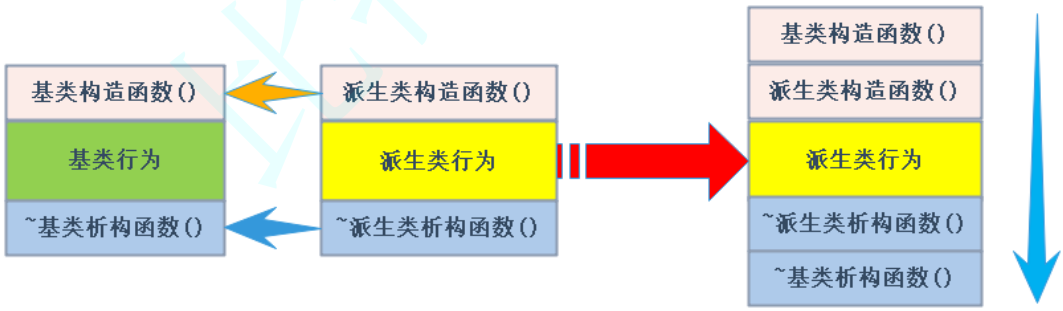
```
{
    B b;
    b.fun(10);
};
```

4.派生类的默认成员函数

6个默认成员函数，**“默认”**的意思就是指我们不写，编译器会变我们自动生成一个，那么在派生类中，这几个成员函数是如何生成的呢？

1. 派生类的构造函数必须调用基类的构造函数初始化基类的那一部分成员。如果基类没有默认的构造函数，则必须在派生类构造函数的初始化列表阶段显示调用。
2. 派生类的拷贝构造函数必须调用基类的拷贝构造完成基类的拷贝初始化。
3. 派生类的operator=必须要调用基类的operator=完成基类的复制。
4. 派生类的析构函数会在被调用完成后自动调用基类的析构函数清理基类成员。因为这样才能保证派生类对象先清理派生类成员再清理基类成员的顺序。
5. 派生类对象初始化先调用基类构造再调派生类构造。
6. 派生类对象析构清理先调用派生类析构再调基类的析构。
7. 因为后续一些场景析构函数需要构成重写，重写的条件之一是函数名相同(这个我们后面会讲解)。那么编译器会对析构函数名进行特殊处理，处理成destrutor()，所以父类析构函数不加virtual的情况下，子类析构函数和父类析构函数构成隐藏关系。






```
class Person
{
public :
    Person(const char* name = "peter")
        : _name(name )
    {
        cout<<"Person()" <<endl;
    }

    Person(const Person& p)
        : _name(p._name)
    {
```

```
        cout<<"Person(const Person& p)" <<endl;
    }

    Person& operator=(const Person& p )
    {
        cout<<"Person operator=(const Person& p)"<< endl;
        if (this != &p)
            _name = p ._name;

        return *this ;
    }

    ~Person()
    {
        cout<<"~Person()" <<endl;
    }
protected :
    string _name ; // 姓名
};

class Student : public Person
{
public :
    Student(const char* name, int num)
        : Person(name )
        , _num(num )
    {
        cout<<"Student()" <<endl;
    }

    Student(const Student& s)
        : Person(s)
        , _num(s ._num)
    {
        cout<<"Student(const Student& s)" <<endl ;
    }

    Student& operator = (const Student& s )
    {
        cout<<"Student& operator= (const Student& s)"<< endl;
        if (this != &s)
        {
            Person::operator =(s);
            _num = s ._num;
        }
        return *this ;
    }

    ~Student()
    {
        cout<<"~Student()" <<endl;
    }
protected :
    int _num ; //学号
};
void Test ()
{
    Student s1 ("jack", 18);
```

```
    Student s2 (s1);
    Student s3 ("rose", 17);
    s1 = s3 ;
}
```

5.继承与友元

> **友元关系不能继承**，也就是说基类友元不能访问子类私有和保护成员

```
class Student;
class Person
{
public:
    friend void Display(const Person& p, const Student& s);
protected:
    string _name; // 姓名
};
class Student : public Person
{
protected:
    int _stuNum; // 学号
};

void Display(const Person& p, const Student& s)
{
    cout << p._name << endl;
    cout << s._stuNum << endl;
}

void main()
{
    Person p;
    Student s;
    Display(p, s);
}
```

6. 继承与静态成员

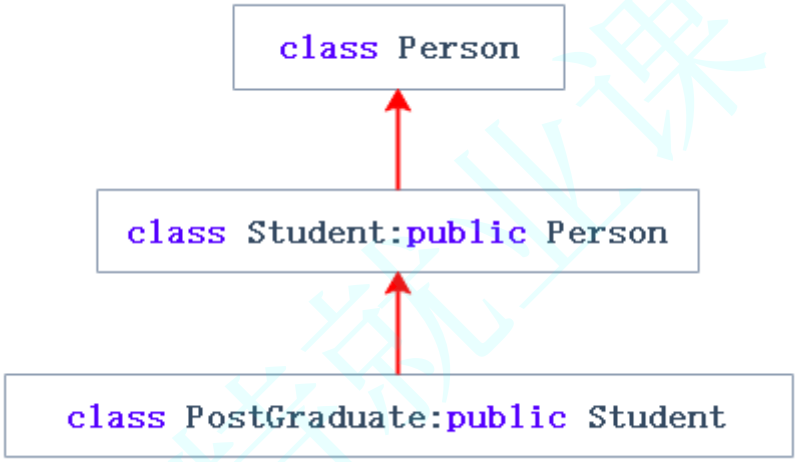
> **基类定义了static静态成员，则整个继承体系里面只有一个这样的成员。**无论派生出多少个子类，都只有一个static成员实例 。

```
class Person
{
public :
    Person () {++ _count ;}
protected :
    string _name ; // 姓名
public :
    static int _count; // 统计人的个数。
};
int Person :: _count = 0;
class Student : public Person
{
protected :
    int _stuNum ; // 学号
};
class Graduate : public Student
```

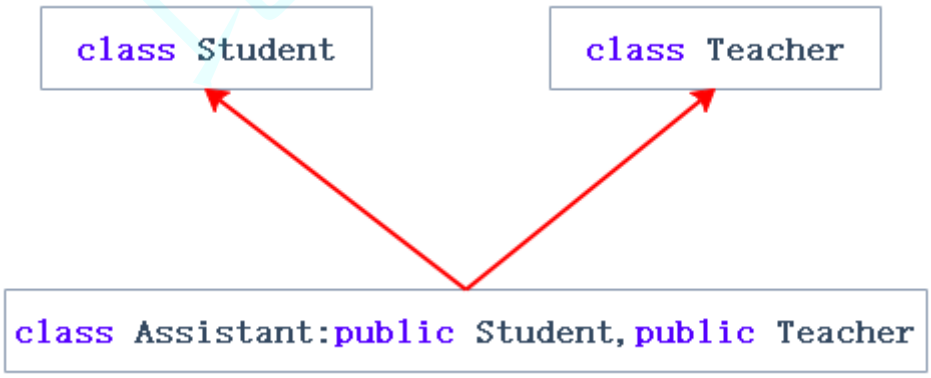
```
{
protected :
    string _seminarCourse ; // 研究科目
};
void TestPerson()
{
    Student s1 ;
    Student s2 ;
    Student s3 ;
    Graduate s4 ;
    cout <<" 人数 :"<< Person ::_count << endl;
    Student ::_count = 0;
    cout <<" 人数 :"<< Person ::_count << endl;
}
```

7.复杂的菱形继承及菱形虚拟继承

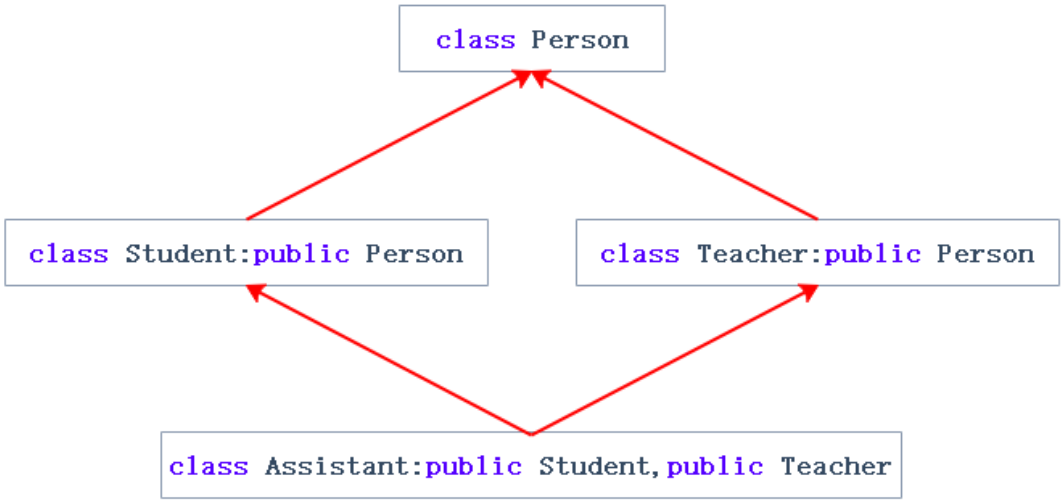
单继承：一个子类只有一个直接父类时称这个继承关系为单继承



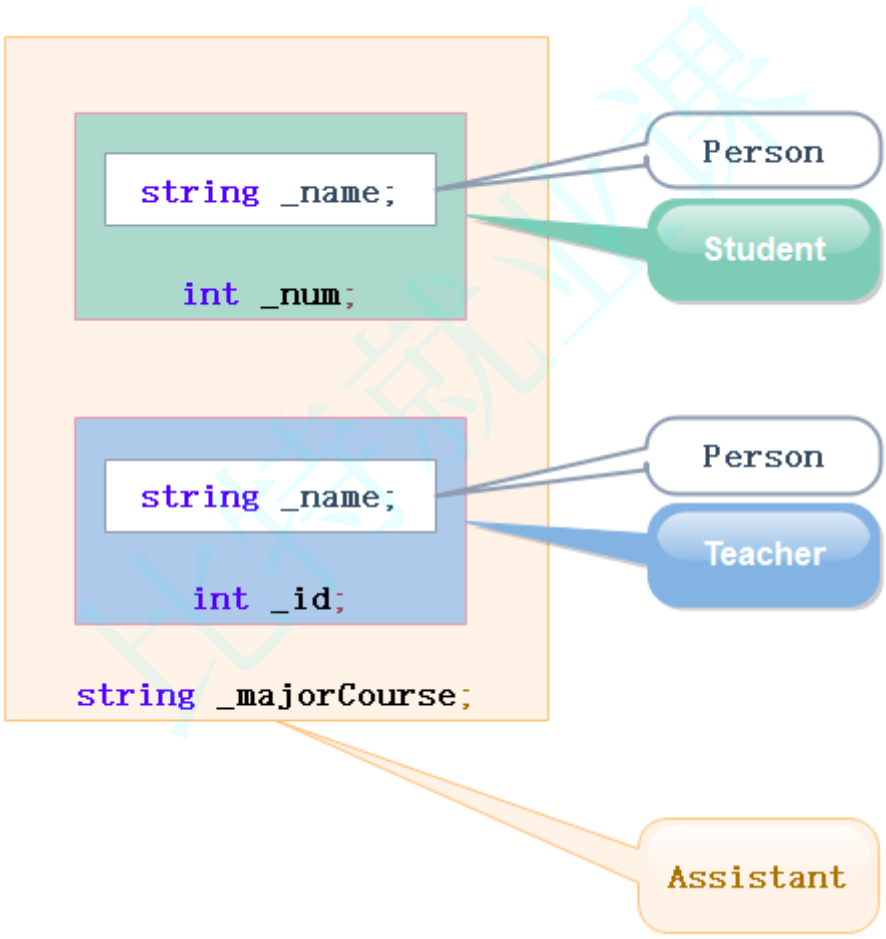
多继承：一个子类有两个或以上直接父类时称这个继承关系为多继承



菱形继承：菱形继承是多继承的一种特殊情况。



菱形继承的问题：从下面的对象成员模型构造，可以看出菱形继承有数据冗余和二义性的问题。在Assistant的对象中Person成员会有两份。




```
class Person
{
public :
    string _name ; // 姓名
};
class Student : public Person
{
protected :
    int _num ; //学号
};
class Teacher : public Person
```

```
{
protected :
    int _id ; // 职工编号
};
class Assistant : public Student, public Teacher
{
protected :
    string _majorCourse ; // 主修课程
};
void Test ()
{
    // 这样会有二义性无法明确知道访问的是哪一个
    Assistant a ;
 a._name = "peter";

 // 需要显示指定访问哪个父类的成员可以解决二义性问题，但是数据冗余问题无法解决
    a.Student::_name = "xxx";
    a.Teacher::_name = "yyy";
}
```

虚拟继承可以解决菱形继承的二义性和数据冗余的问题。如上面的继承关系，在Student和Teacher的继承Person时使用虚拟继承，即可解决问题。需要注意的是，虚拟继承不要在其他地方去使用。

```
class Person
{
public :
    string _name ; // 姓名
};
class Student : virtual public Person
{
protected :
    int _num ; //学号
};
 class Teacher : virtual public Person
{
protected :
    int _id ; // 职工编号
};
class Assistant : public Student, public Teacher
{
protected :
    string _majorCourse ; // 主修课程
};
void Test ()
{
    Assistant a ;
    a._name = "peter";
}
```

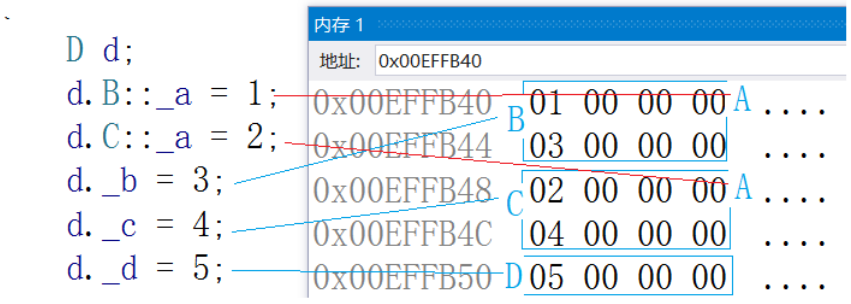
虚拟继承解决数据冗余和二义性的原理

为了研究虚拟继承原理，我们给出了一个简化的菱形继承继承体系，再借助**内存窗口观察对象成员的模型。**

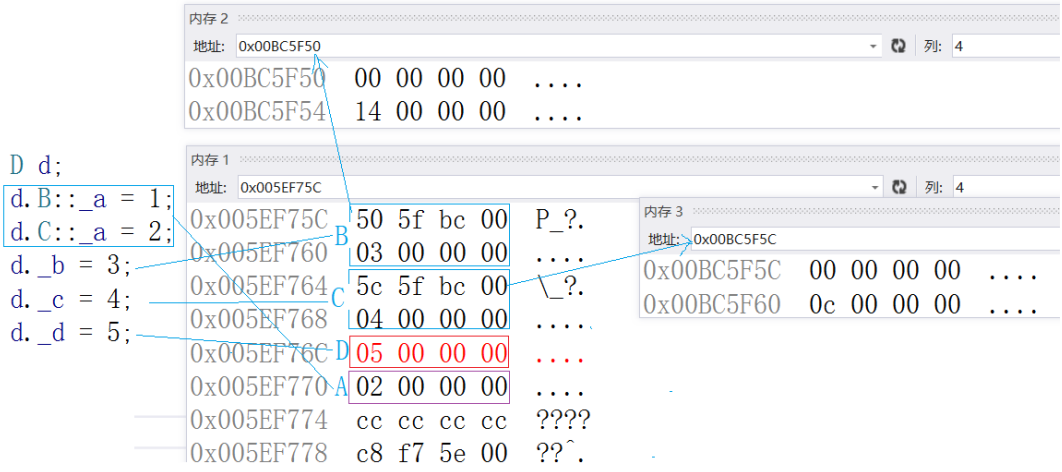
```
class A
{
```

```
public:
    int _a;
};

// class B : public A
class B : virtual public A
{
public:
    int _b;
};

// class C : public A
class C : virtual public A
{
public:
    int _c;
};

class D : public B, public C
{
public:
    int _d;
};

int main()
{
    D d;
    d.B::_a = 1;
    d.C::_a = 2;
    d._b = 3;
    d._c = 4;
    d._d = 5;

    return 0;
}
```

下图是菱形继承的内存对象成员模型：这里可以看到数据冗余

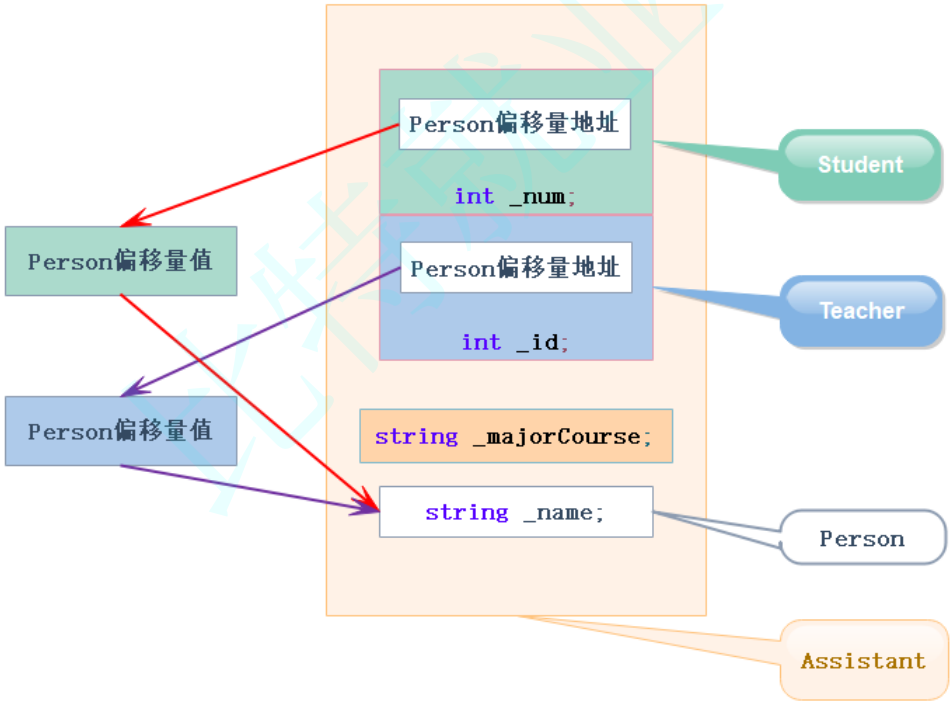


下图是菱形虚拟继承的内存对象成员模型：这里可以分析出D对象中将A放到的了对象组成的最下面，这个A同时属于B和C，那么B和C如何去找到公共的A呢？**这里是通过了B和C的两个指针，指向的一张表。这两个指针叫虚基表指针，这两个表叫虚基表。虚基表中存的偏移量。通过偏移量可以找到下面的A。**




```
// 有童鞋会有疑问为什么D中B和C部分要去找属于自己的A？那么大家看看当下面的赋值发生时，d是不是要去找出B/C成员中的A才能赋值过去？
    D d;
B b = d;
C c = d;
```

下面是上面的Person关系菱形虚拟继承的原理解释：



8.继承的总结和反思

1. **很多人说C++语法复杂，其实多继承就是一个体现。有了多继承**，就存在菱形继承，有了菱形继承就有菱形虚拟继承，底层实现就很复杂。所以一般不建议设计出多继承，一定不要设计出菱形继承。否则在复杂度及性能上都有问题。
2. 多继承可以认为是C++的缺陷之一，很多后来的OO语言都没有多继承，如Java。
3. **继承和组合**
    - public继承是一种**is-a**的关系。也就是说每个派生类对象都是一个基类对象。
    - 组合是一种**has-a**的关系。假设B组合了A，每个B对象中都有一个A对象。

- 优先使用对象组合，而不是类继承。
- 继承允许你根据基类的实现来定义派生类的实现。这种通过生成派生类的复用通常被称为白箱复用(white-box reuse)。术语“白箱”是相对可视性而言：在继承方式中，基类的内部细节对子类可见 。继承一定程度破坏了基类的封装，基类的改变，对派生类有很大的影响。派生类和基类间的依赖关系很强，耦合度高。
- 对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组装或组合对象来获得。对象组合要求被组合的对象具有良好定义的接口。这种复用风格被称为黑箱复用(black-box reuse)，因为对象的内部细节是不可见的。对象只以“黑箱”的形式出现。组合类之间没有很强的依赖关系，耦合度低。优先使用对象组合有助于你保持每个类被封装。
- 实际尽量多去用组合。组合的耦合度低，代码维护性好。不过继承也有用武之地的，有些关系就适合继承那就用继承，另外要实现多态，也必须要继承。类之间的关系可以用继承，可以用组合，就用组合。

```
// Car和BMW Car和Benz构成is-a的关系
  class Car{
  protected:
   string _colour = "白色";  // 颜色
   string _num = "陕ABIT00";    // 车牌号
  };

  class BMW : public Car{
  public:
   void Drive() {cout << "好开-操控" << endl;}
  };

  class Benz : public Car{
  public:
   void Drive() {cout << "好坐-舒适" << endl;}
  };

  // Tire和Car构成has-a的关系

  class Tire{
  protected:
      string _brand = "Michelin";  // 品牌
      size_t _size = 17;             // 尺寸

  };

  class Car{
  protected:
   string _colour = "白色";       // 颜色
   string _num = "陕ABIT00";      // 车牌号
   Tire _t;                      // 轮胎
  };
```

9.笔试面试题

1. 什么是菱形继承？菱形继承的问题是什么？
2. 什么是菱形虚拟继承？如何解决数据冗余和二义性的
3. 继承和组合的区别？什么时候用继承？什么时候用组合？