继承复习

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，它允许我们在保持原有类特性的基础上进行扩展，增加方法(成员函数)和属性(成员变量)，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。以前我们接触的函数层次的复用，继承是类设计层次的复用。

下面我们看到没有继承之前我们设计了两个类Student和Teacher，Student和Teacher都有姓名/地址/电话/年龄等成员变量，都有identity身份认证的成员函数，设计到两个类里面就是冗余的。当然他们也有一些不同的成员变量和函数，比如老师独有成员变量是职称，学生的独有成员变量是学号；学生的独有成员函数是学习，老师的独有成员函数是授课

代码块

```
class Student
{
public:
    // 进入校园/图书馆/实验室刷二维码等身份认证
    void identity()
    {
    // ...
    }

    // 学习
    void study()
    {
    // ...
    }

protected:
    string _name = "peter"; // 姓名
    string _address; // 地址
    string _tel; // 电话
    int _age = 18; // 年龄
    int _stuid; // 学号
};

class Teacher
{
```

```
public:
    // 进入校园/图书馆/实验室刷二维码等身份认证
    void identity()
    {
    // ...
    }
    // 授课
    void teaching()
    {
    //...
    }

protected:
    string _name = "张三"; // 姓名
    int _age = 18; // 年龄
    string _address; // 地址
    string _tel; // 电话
    string _title; // 职称
};

int main()
{
    return 0;
}

```

下面我们公共的成员都放到Person类中，Student和teacher都继承Person，就可以复用这些成员，就不需要重复定义了，省去了很多麻烦。

代码块

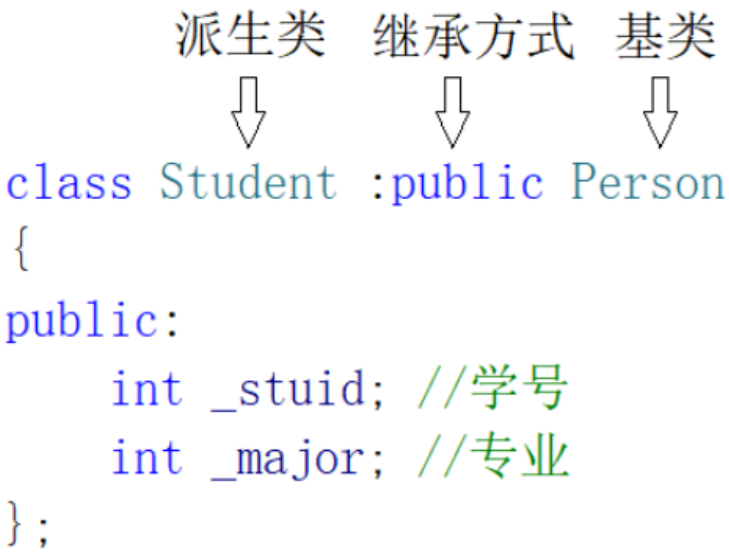
```
class Person
{
public:
    // 进入校园/图书馆/实验室刷二维码等身份认证
    void identity()
    {
        cout << "void identity()" <<_name<< endl;
    }

protected:
    string _name = "张三"; // 姓名
    string _address; // 地址
    string _tel; // 电话
    int _age = 18; // 年龄
};
```

```cpp

class Student : public Person
{
public:
    // 学习
    void study()
    {
        // ...
    }

protected:
    int _stuid; // 学号
};

class Teacher : public Person
{
public:
    // 授课
    void teaching()
    {
        //...
    }

protected:
    string title; // 职称
};

int main()
{
    Student s;
    Teacher t;
    s.identity();
    t.identity();
    return 0;
}

```

1.2 继承的定义

1.2.1 定义格式

> 下面我们看到Person是基类，也称作父类。Student是派生类，也称作子类。(因为翻译的原因，所以既叫基类/派生类，也叫父类/子类)



1.2.2 继承的访问控制

a. **public继承（最常用）**
   - 基类的public成员在派生类中仍为public
   - 基类的protected成员在派生类中仍为protected
   - 基类的private成员在派生类中不可访问

b. **protected继承**
   - 基类的public和protected成员在派生类中都变为protected
   - 基类的private成员在派生类中不可访问

c. **private继承（默认方式，但不常用）**
   - 基类的public和protected成员在派生类中都变为private
   - 基类的private成员在派生类中不可访问

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

a. 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。

b. 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。

c. 实际上面的表格我们进行一下总结会发现，基类的私有成员在派生类都是不可见。基类的其他成员在派生类的访问方式 == Min(成员在基类的访问限定符，继承方式)，public > protected > private。

d. 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。

e. 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。

代码块

```
// 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
```

```
21   };
22
```

1.3 继承类模版

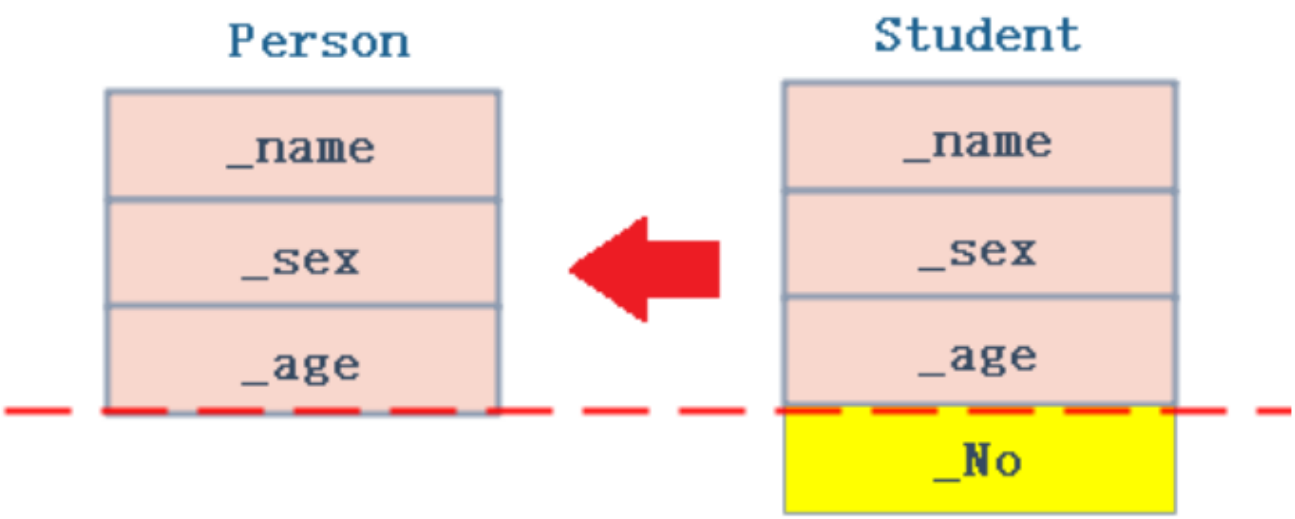
代码块

```
1   namespace bit
2   {
3       //template<class T>
4       //class vector
5       //{};
6       // stack和vector的关系，既符合is-a，也符合has-a
7       template<class T>
8       class stack : public std::vector<T>
9       {
10      public:
11          void push(const T& x)
12          {
13              // 基类是类模板时，需要指定一下类域，
14              // 否则编译报错:error C3861: “push_back”: 找不到标识符
15              // 因为stack<int>实例化时，也实例化vector<int>了
16              // 但是模版是按需实例化，push_back等成员函数未实例化，所以找不到
17              vector<T>::push_back(x);
18              //push_back(x);
19          }
20          void pop() { vector<T>::pop_back(); }
21          const T& top() { return vector<T>::back(); }
22          bool empty() { return vector<T>::empty(); }
23      };
24  }
25
26  int main()
27  {
28      bit::stack<int> st;
29      st.push(1);
30      st.push(2);
31      st.push(3);
32      while (!st.empty())
33      {
34      cout << st.top() << " ";
35      st.pop();
36      }
37      return 0;
```

```
38  }
39
```

2. 基类和派生类之间的转换

- public继承的派生类对象 可以赋值给 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。寓意把派生类中基类那部分切出来，基类指针或引用指向的是派生类中切出来的基类那部分。
- 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。



代码块

```
1   class Person
2   {
3   protected:
4       string _name; // 姓名
5       string _sex; // 性别
6       int _age; // 年龄
7   };
8
9   class Student : public Person
10  {
11  public :
12      int _No ; // 学号
```

```cpp
};

int main()
{
    Student sobj ;
    // 1.派生类对象可以赋值给基类的指针/引用
    Person* pp = &sobj;
    Person& rp = sobj;

    // 生类对象可以赋值给基类的对象是通过调用后面会讲解的基类的拷贝构造完成的
    Person pobj = sobj;

    //2.基类对象不能赋值给派生类对象，这里会编译报错
    sobj = pobj;

    return 0;
}
```

3. 继承中的作用域

3.1 隐藏规则

a. 在继承体系中基类和派生类都有独立的作用域。

b. 派生类和基类中有同名成员，派生类成员将屏蔽基类对同名成员的直接访问，这种情况叫隐藏。（在派生类成员函数中，可以使用基类::基类成员 显示访问）

c. 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。

d. 注意在实际中在继承体系里面最好不要定义同名的成员。

代码块

```cpp
// 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
};

int main()
{
    Student s1;
    s1.Print();

    return 0;
};

```

3.2 考察继承作用域

代码块

```
class A
{
public:
    void fun()
    {
        cout << "func()" << endl;
    }
};

class B : public A
{
public:
    // 可以在这里加上 "using A::fun;" 避免隐藏
    void fun(int i)
    {
        cout << "func(int i)" <<i<<endl;
    }
};

int main()
```

```
21  {
22      B b;
23      b.fun(10);
24      b.fun();      // <- B中继承的A的fun被隐藏,报错函数传参太少
25      return 0;
26  };
27
```

4. 派生类的默认成员函数

4.1 4个常见的默认成员函数

a. 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。

b. 派生类的拷贝构造函数必须调用基类的拷贝构造完成对基类的拷贝初始化。

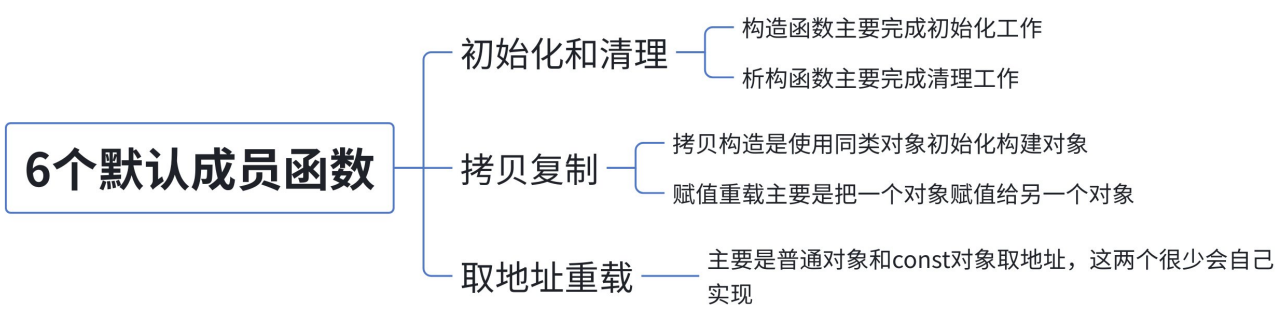
c. 派生来的operator=必须要调用基类的operator=完成基类的复制。需要注意的是派生类的operator=隐藏了基类的operator=，所以显示调用基类的operator=，需要指定基类作用域。

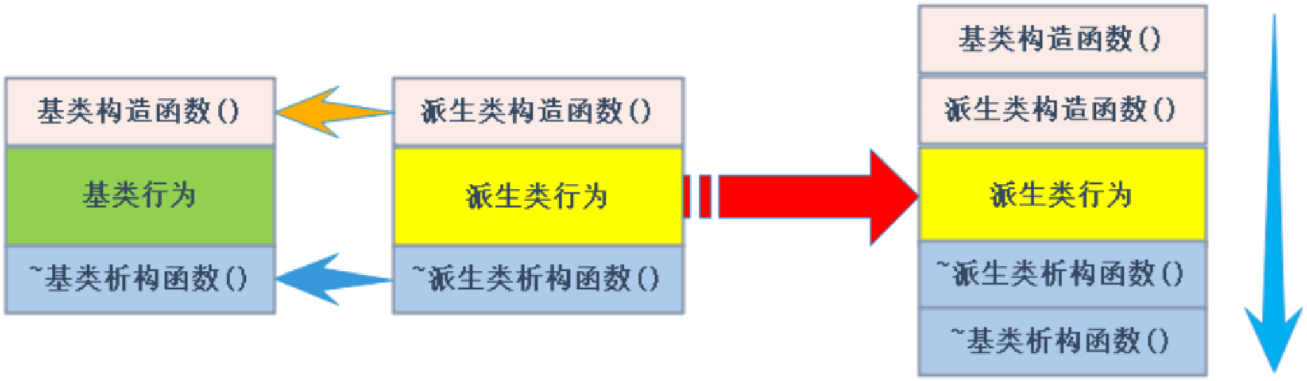
d. 派生类的析构函数会在调用完成后自动调用基类的析构函数清理基类成员。因为这样才能保证派生类对象先清理派生类成员再清理基类成员的顺序。

e. 派生类对象初始化先调用基类构造函数再调用派生类构造。

f. 派生类对象析构清理先调用派生类析构再调用基类的析构。

g. 因为多态中的一些场景，析构函数需要构成重写，重写的条件之一是函数名相同。那么编译器会对析构函数名进行特殊处理，处理成 `destructor()` ，所以基类析构函数不加virtual的情况下，派生类析构函数和基类析构函数构成隐藏关系。





代码块

```
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
{
```

```cpp
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)
        {
                // 构成隐藏，所以需要显示调用
                Person::operator =(s);
                _num = s._num;
        }
        return *this;
    }

    ~Student()
    {
        cout << "~Student()" << endl;
    }

protected:
    int _num; //学号
};

int main()
{
    Student s1("jack", 18);
    Student s2(s1);
    Student s3("rose", 17);
    s1 = s3;
    return 0;
}

```

4.2 实现一个不能被继承的类

- **方法1：**基类构造函数私有，派生类的构造必须调用基类的构造函数，但是街垒的构造函数私有化以后，派生类看不到基类构造函数就不能调用了，那么派生类就无法实例化出对象。
- **方法2：** C++11新增了一个关键字，final修改基类，派生类就不能继承了。

代码块

```
// C++11的方法
class Base final
{
public:
    void func5() { cout << "Base::func5" << endl; }

protected:
    int a = 1;

private:
    // C++98的方法
    /*Base()
    {}*/
};

class Derive :public Base
{
    void func4() { cout << "Derive::func4" << endl; }
protected:
    int b = 2;
};

int main()
{
    Base b;
    Derive d;
    return 0;
}
```

5. 继承与友元

友元关系不能被继承，也就说基类友元不能访问派生类私有和保护成员。

```cpp
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;
}

int main()
{
    Person p;
    Student s;

    // 编译报错: error C2248: “Student::_stuNum”: 无法访问 protected 成员
    // 解决方案: Display也变成Student 的友元即可
    Display(p, s);

    return 0;
}
```

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。无论派生出多少个派生类，都只有一个static成员实例。

代码块

```cpp
class Person
{
```

```cpp
public:
    string _name;
    static int _count;
};

int Person::_count = 0;

class Student : public Person
{
protected:
    int _stuNum;
};

int main()
{
    Person p;
    Student s;

    // 这里的运行结果可以看到非静态成员_name的地址是不一样的
    // 说明派生类继承下来了，父派生类对象各有一份
    cout << &p._name << endl;
    cout << &s._name << endl;

    // 这里的运行结果可以看到静态成员_count的地址是一样的
    // 说明派生类和基类共用同一份静态成员
    cout << &p._count << endl;
    cout << &s._count << endl;

    // 公有的情况下，父派生类指定类域都可以访问静态成员
    cout << Person::_count << endl;
    cout << Student::_count << endl;

    return 0;
}
```

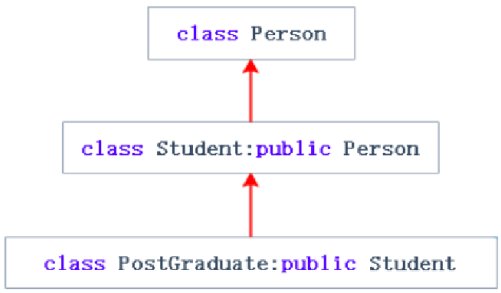
7. 多继承及其菱形继承问题

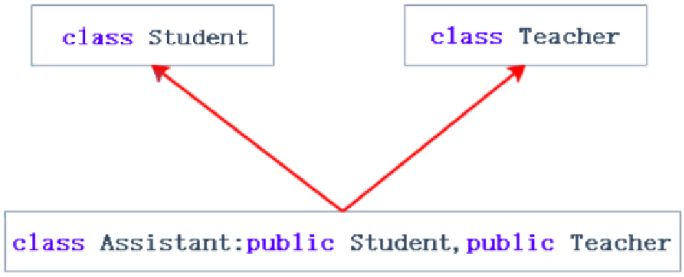
7.1 继承模型

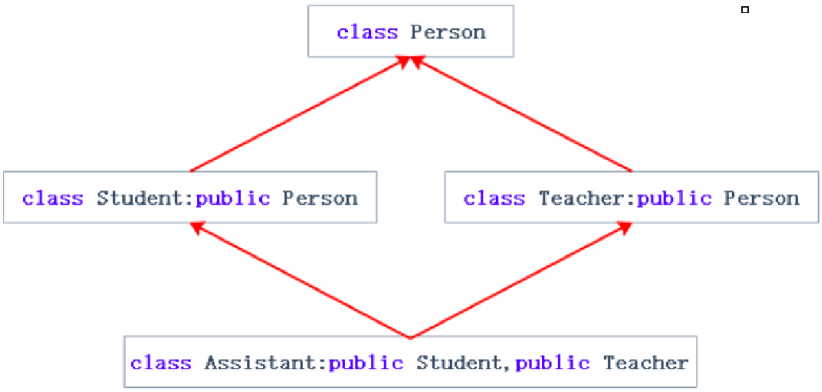
单继承： 一个派生类只有一个直接基类。

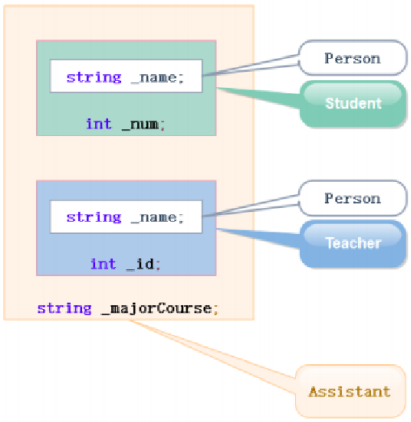
多继承： 一个派生类有两个或以上直接基类时称这个继承关系为多继承，多继承对象在内存的模型是：先继承的基类在前面，后面继承的基类在后面，派生类成员放到最后面。

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。支出多继承就一定会有菱形继承。Java就直接不支持多继承，规避掉了这个问题，所以实践中也不建议设计出菱形继承这样的模型的。









代码块

```
1
2   class Person
3   {
4   public:
5       string _name; // 姓名
6   };
7
8   class Student : public Person
9   {
10  protected:
11      int _num; //学号
12  };
13
14  class Teacher : public Person
15  {
16  protected:
17      int _id; // 职工编号
18  };
```

```cpp

class Assistant : public Student, public Teacher
{
protected:
    string _majorCourse; // 主修课程
};

int main()
{
    // 编译报错: error C2385: 对“_name”的访问不明确
    Assistant a;
    a._name = "peter";

    // 需要显示指定访问哪个基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";

    return 0;
}
```

7.2 虚继承

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，性能也会有一些损失，所以最好不要设计出菱形继承。多继承可以认为是C++的缺陷之一，后来的一些编程语言都没有多继承，如Java。

代码块

```cpp
class Person
{
public:
    string _name; // 姓名
    /*int _tel;
    int _age;
    string _gender;
    string _address;*/
    // ...
};

// 使用虚继承Person类
class Student : virtual public Person
{
protected:
    int _num; //学号
```

```
};

// 使用虚继承Person类
class Teacher : virtual public Person
{
protected:
    int _id; // 职工编号
};

// 教授助理
class Assistant : public Student, public Teacher
{
protected:
    string _majorCourse; // 主修课程
};

int main()
{
    // 使用虚继承，可以解决数据冗余和二义性
    Assistant a;
    a._name = "peter";
    return 0;
}
```

我们可以设计出多继承，但是不建议设计出菱形继承，因为菱形虚拟继承以后，无论是使用还是底层都会复杂很多。当然有多继承语法支持，就一定存在会设计出菱形继承。

代码块

```
class Person
{
public:
    Person(const char* name)
            :_name(name)
    {}
    string _name; // 姓名
};

class Student : virtual public Person
{
public:
    Student(const char* name, int num)
        :Person(name)
        , _num(num)
    {}

```

```
protected:
    int _num; //学号
};

class Teacher : virtual public Person
{
public:
    Teacher(const char* name, int id)
        :Person(name)
        , _id(id)
    {}

protected:
    int _id; // 职工编号
};

// 不要去玩菱形继承
class Assistant : public Student, public Teacher
{
public:
    Assistant(const char* name1, const char* name2, const char* name3)
        :Person(name3)
        , Student(name1, 1)
        , Teacher(name2, 2)
    {}

protected:
    string _majorCourse; // 主修课程
};

int main()
{
    // 思考一下这里a对象中_name是"张三", "李四", "王五"中的哪一个?
    Assistant a("张三", "李四", "王五");

    return 0;
}
```

a对象中所继承的Person中的 `_name` 就是它自己默认构造中传的参，也就是 `name3` ，而不由其派生类中的构造函数对其构造函数调用时传的参数。

7.3 多继承中指针偏移问题

代码块

```cpp
class Base1 { public: int _b1; };
class Base2 { public: int _b2; };
class Derive : public Base1, public Base2 { public: int _d; };

int main()
{
    Derive d;
    Base1* p1 = &d;
    Base2* p2 = &d;
    Derive* p3 = &d;

    cout << &d << endl;
    cout << (p1) << endl;
    cout << (p2) << endl;
    cout << (p3) << endl;

    return 0;
}
```

代码块

```
//输出结果:
00000076B950FC48
00000076B950FC48
00000076B950FC4C
00000076B950FC48
```

原因：

a. 对象内存布局

- `Derive` 继承自 `Base1` 和 `Base2`，内存布局依次为：

  代码块

  ```
  [Base1::_b1][Base2::_b2][Derive::_d]
  ```

- `Base1` 子对象位于起始位置，`Base2` 子对象紧随其后（偏移 `sizeof(Base1)` 字节）。

b. 指针转换的地址调整

- `Base1* p1 = &d`：
  `Base1` 是第一个基类，无需偏移，`p1` 直接指向 `Derive` 对象的起始地址（与 `&d` 相同）。

- `Base2* p2 = &d`：
  `Base2` 是第二个基类，编译器自动将地址偏移 `sizeof(Base1)`（即 `4` 字节，假设 `int` 占 `4` 字节），因此 `p2` 的地址比 `&d` 大 `4`。
- `Derive* p3 = &d`：
  原始指针无需调整，与 `&d` 地址相同。

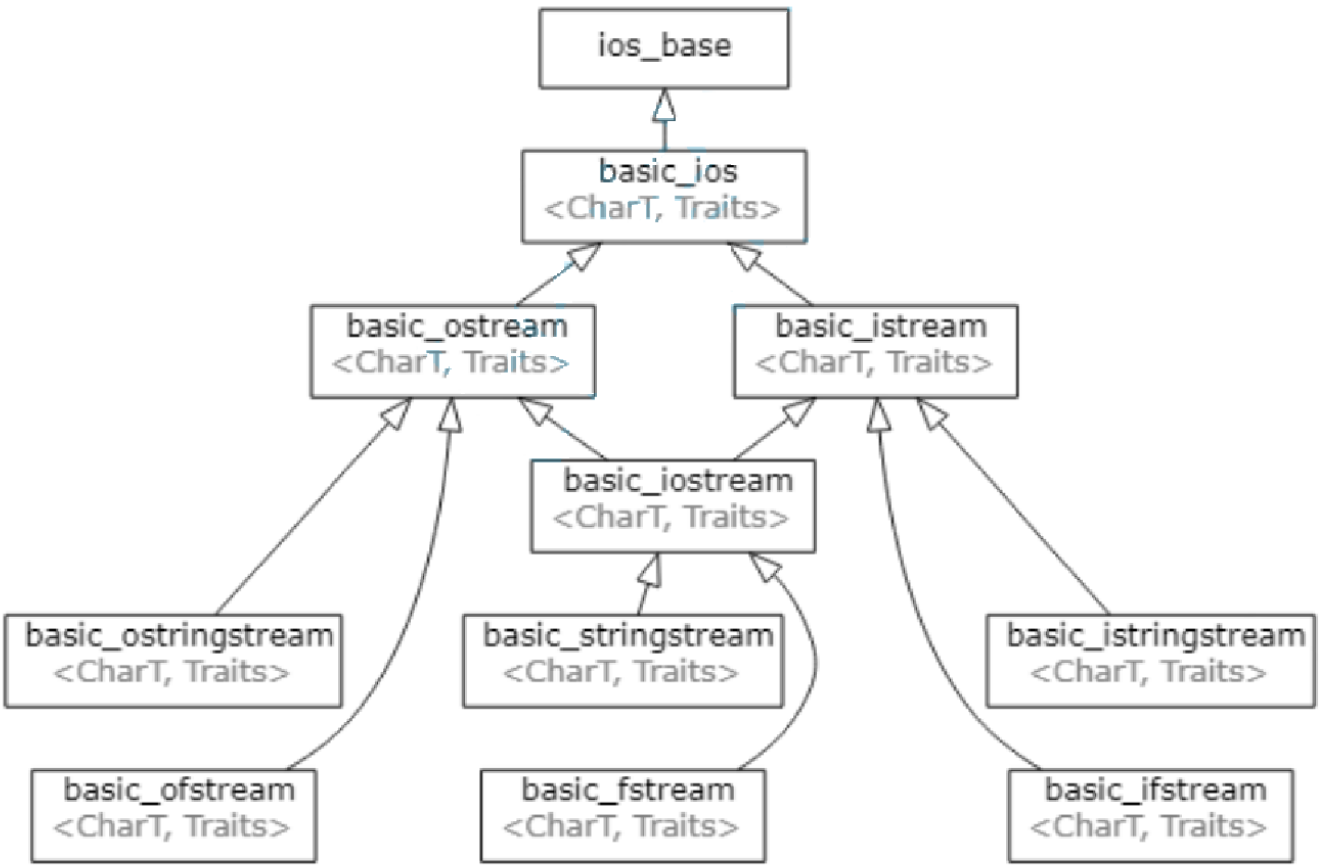
c. 输出结果

- 地址值可能因系统而异，但相对关系固定：
  `p1 == p3 == &d`，而 `p2` 比它们大 `sizeof(int)`。

关键点：

- 多继承时，派生类对象包含多个基类子对象，布局按声明顺序排列。
- 基类指针转换时，编译器自动处理地址偏移，确保指向正确的子对象部分。
- `cout` 直接输出指针时，打印的是它们指向的实际内存地址。

7.4 IO库中的菱形虚拟继承



代码块

```
1   template<class CharT, class Traits = std::char_traits<CharT>>
```

```
class basic_ostream : virtual public std::basic_ios<CharT, Traits>
{};

template<class CharT, class Traits = std::char_traits<CharT>>
class basic_istream : virtual public std::basic_ios<CharT, Traits>
{};
```

8. 继承和组合

8.1 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。
- 继承允许你根据基类的实现来定哟派生类的实现。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，基类的内部细节对派生类可见。继承一定程度的破换了基类的封装，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。
- 对象组合时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。对象足额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保持每个类的封装。
- 优先使用组合，而不是继承。实际尽量多用组合，组合耦合度更低，代码维护性好。不过也不是那么绝对，类之间的关系只适合继承(is-a)，那就用继承，另外要实现多态，也必须要求继承。类之间的关系既适合用继承(is-a)也适合用组合(has-a)，就用组合。

代码块

```
// Tire(轮胎)和Car(车)更符合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 _t1; // 轮胎
```

```
    Tire _t2; // 轮胎
    Tire _t3; // 轮胎
    Tire _t4; // 轮胎
};

class BMW : public Car {

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

// Car和BMW/Benz更符合is-a的关系
class Benz : public Car
{
public:
    void Drive() { cout << "好坐-舒适" << endl; }
};

template<class T>
class vector
{
};

// stack和vector的关系，既符合is-a，也符合has-a
template<class T>
class stack : public vector<T>
{
};

template<class T>
class stack
{
public:
    vector<T> _v;
};

int main()
{
    return 0;
}

```