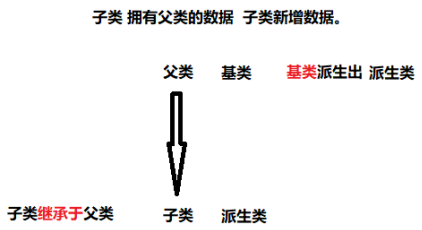
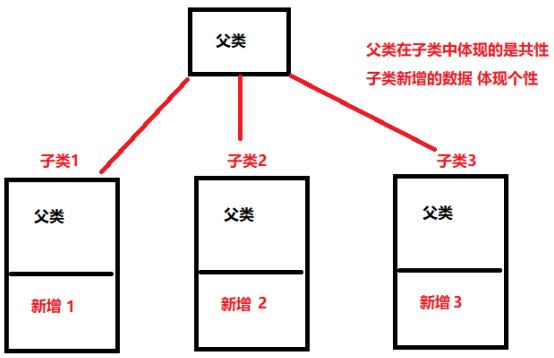
知识点1【继承概述】

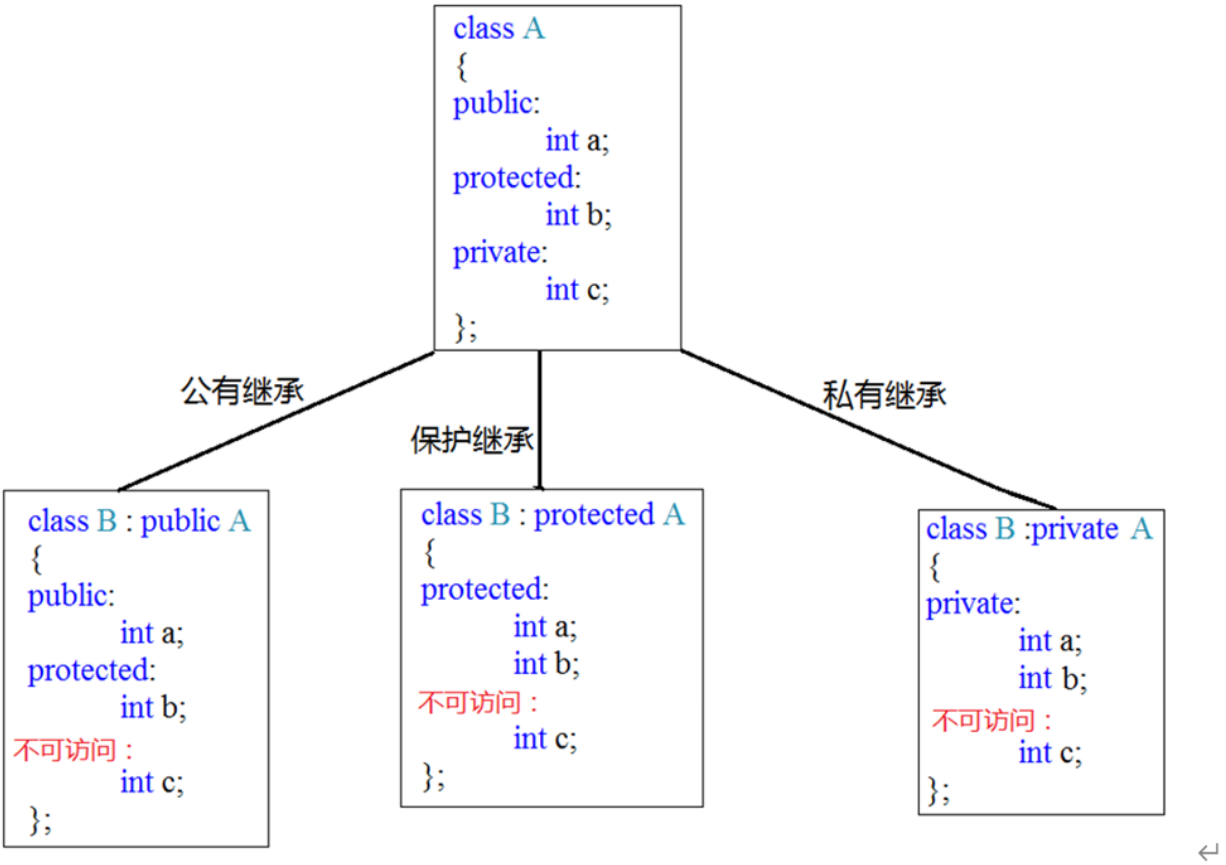




知识点2【子类的定义形式】

```
class 父类{};
class 子类:继承方式 父类名
{
 //新增子类数据
};
```

继承方式：private protected public(推荐)

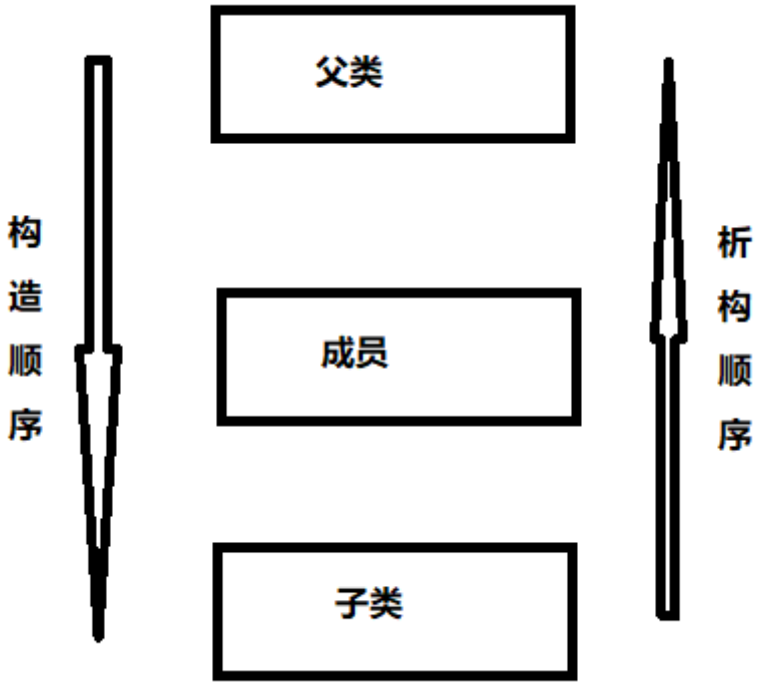


公共继承 保持不变，保护继承变保护，私有继承变私有，所有父类私有在子类中不可见。

```
class Base
{
private:
 int a;
protected:
 int b;
public:
 int c;
};
 class Son:public Base
 {
  //子类中 能访问 protected b public c
```

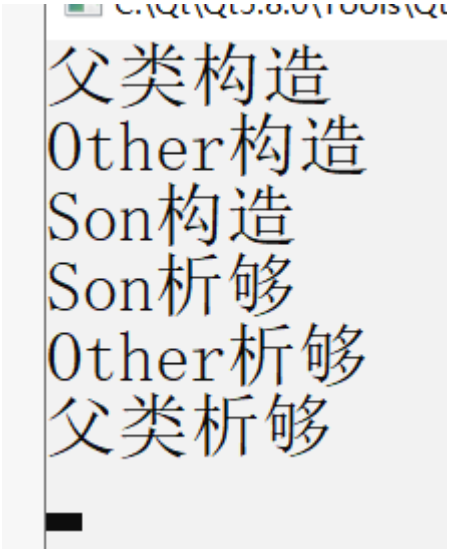
```
public:
 void func(void)
 {
 cout<<b<<c<<endl;
 //cout<<a<<endl;//不可访问
 }
};

void test01()
{
 Son ob;
 //cout<<ob.b<<endl;//类外无法访问
 cout<<ob.c<<endl;
}
```

知识点3【子类的构造析够顺序】




```
class Base
{
public:
 Base()
 {
 cout<<"父类构造"<<endl;
 }
```

```
8  ~Base()
9  {
10  cout<<"父类析够"<<endl;
11  }
12 };
13 class Other
14 {
15 public:
16  Other()
17  {
18  cout<<"Other构造"<<endl;
19  }
20  ~Other()
21  {
22  cout<<"Other析够"<<endl;
23  }
24 };
25
26 class Son:public Base
27 {
28 public:
29  Other ob;
30 public:
31 public:
32  Son()
33  {
34  cout<<"Son构造"<<endl;
35  }
36  ~Son()
37  {
38  cout<<"Son析够"<<endl;
39  }
40 };
41
42 void test01()
43 {
44  Son ob;
45 }
```



知识点4【子类调用成员对象、父类的有参构造】

子类 会自动调用 成员对象、父类的默认构造。

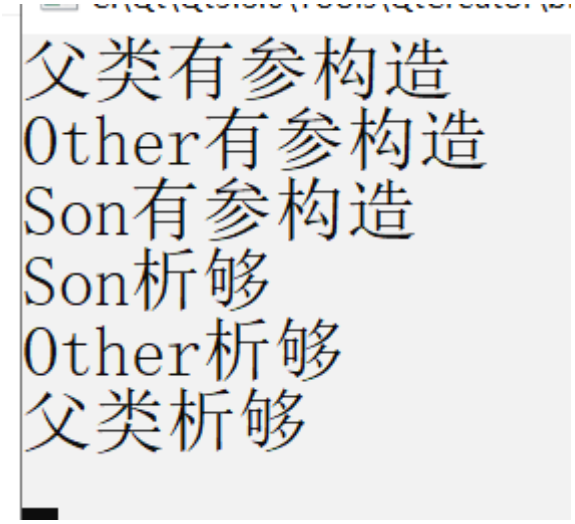
子类 必须使用初始化列表 调用成员对象、父类的有参构造。

初始化列表时：父类写类名称 成员对象用对象名。

```
#include <iostream>

using namespace std;
class Base
{
public:
 int a;
public:
 Base()
 {
 cout<<"父类默认构造"<<endl;
 }
 Base(int a)
 {
 this->a = a;
 cout<<"父类有参构造"<<endl;
 }
 ~Base()
 {
 cout<<"父类析够"<<endl;
 }
};
class Other
```

```
{
public:
 int b;
public:
 Other()
 {
 cout<<"Other默认构造"<<endl;
 }
 Other(int b)
 {
 this->b = b;
 cout<<"Other有参构造"<<endl;
 }
 ~Other()
 {
 cout<<"Other析够"<<endl;
 }
};

class Son:public Base
{
public:
 Other ob;
 int c;
public:
 Son()
 {
 cout<<"Son构造"<<endl;
 }
 //父类写类名称 成员对象用对象名
 Son(int a, int b, int c):Base(a), ob(b)
 {
 this->c = c;
 cout<<"Son有参构造"<<endl;
 }
 ~Son()
 {
 cout<<"Son析够"<<endl;
 }
};
```

```
64
65 void test01()
66 {
67  Son ob(10,20,30);
68 }
69
70 int main(int argc, char *argv[])
71 {
72  test01();
73  return 0;
74 }
75
```



知识点5【子类和父类同名成员处理】

同名成员 最简单 最安全的处理方式：加作用域

1、子类和父类 同名成员数据

子类默认优先访问 子类的同名成员

必须加父类作用域 访问父类的同名成员

```
1 class Base
2 {
3 public:
4  int a;
5 public:
6  Base(int a)
7  {
8  this->a = a;
9  }
10 };
```

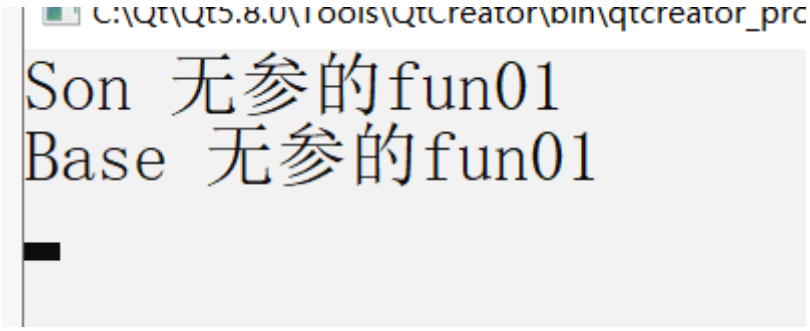
```
11
12 class Son:public Base
13 {
14 public:
15  int a;
16  Son(int x, int y):Base(x)
17  {
18  a = y;
19  }
20 };
21
22 void test01()
23 {
24  Son ob(10,20);
25  //子类默认优先访问 子类的同名成员
26  cout<<ob.a<<endl;
27  //必须加父类作用域 访问父类的同名成员
28  cout<<ob.Base::a<<endl;
29 }
```

```
20
10
```

2、子类和父类 同名成员函数

```
1 class Base
2 {
3 public:
4  void fun01()
5  {
6  cout<<"Base 无参的fun01"<<endl;
7  }
8 };
9
10 class Son:public Base
11 {
12 public:
13  void fun01()
14  {
15  cout<<"Son 无参的fun01"<<endl;
```

```
 }
};

void test01()
{
 Son ob;
 //子类默认优先访问 子类的同名成员
 ob.fun01();
 //必须加父类作用域 访问父类的同名成员
 ob.Base::fun01();
}
```



3、子类 重定义 父类的同名函数

重载：无继承，同一作用域，参数的个数、顺序、类型不同 都可重载

重定义：有继承， 子类 重定义 父类的同名函数（参数可以不同）（非虚函数）

子类一旦 重定义了父类的同名函数（不管参数是否一致），子类中都将屏蔽父类所有的同名函数。

```
class Base
{
public:
 void fun01()
 {
 cout<<"Base 无参的fun01"<<endl;
 }
 void fun01(int a)
 {
  cout<<"Base 的fun01 int"<<endl;
 }
 void fun01(int a, int b)
 {
 cout<<"Base 的fun01 int int"<<endl;
 }
```

```
16 };
17
18 class Son:public Base
19 {
20 public:
21  void fun01(string a)
22  {
23  cout<<"Son 的fun01 char"<<endl;
24  }
25 };
26
27 void test01()
28 {
29  Son ob;
30  ob.fun01("hello");
31  ob.fun01();
32  ob.fun01(10);
33  ob.fun01(10,20);
34 }
```

```
30 void test01()
31 {
32     Son ob;
33     ob.fun01("hello");
34     ob.fun01();
35     ob.fun01(10);
36     ob.fun01(10,20);
37
38 }
```

不识别父类的同名函

需要将父类的作用域 才能识别 屏蔽的函数

```
void test01()
{
    Son ob;
    ob.fun01("hello");
    ob.Base::fun01();
    ob.Base::fun01(10);
    ob.Base::fun01(10,20);

}
```

```
C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.ex
Son 的fun01 char
Base 无参的fun01
Base 的fun01 int
Base 的fun01 int int
```

知识点6【子类不能继承父类成员】

父类的构造、析够、operator= 不能被子类 继承

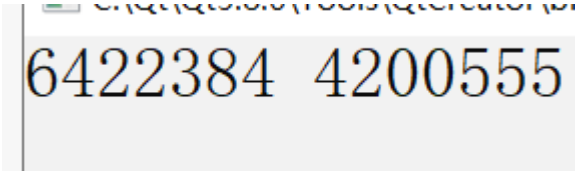
知识点7【多继承】（了解）

1、多继承的格式

```
class 父类1{};
class 父类2{};


class 子类:继承方式1 父类1, 继承方式2 父类2
{
 //新增子类数据
};
```

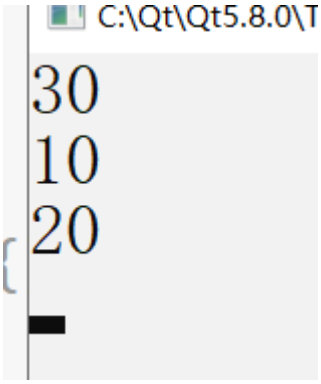
子类就拥有了父类1，父类2的大部分数据

```
class Base1
{
public:
 int a;
};
class Base2
{
public:
 int b;
};


class Son:public Base1, public Base2
{
public:

};

void test01()
{
 Son ob;
 cout<<ob.a<<" "<<ob.b<<endl;

}
```



2、多继承中同名成员

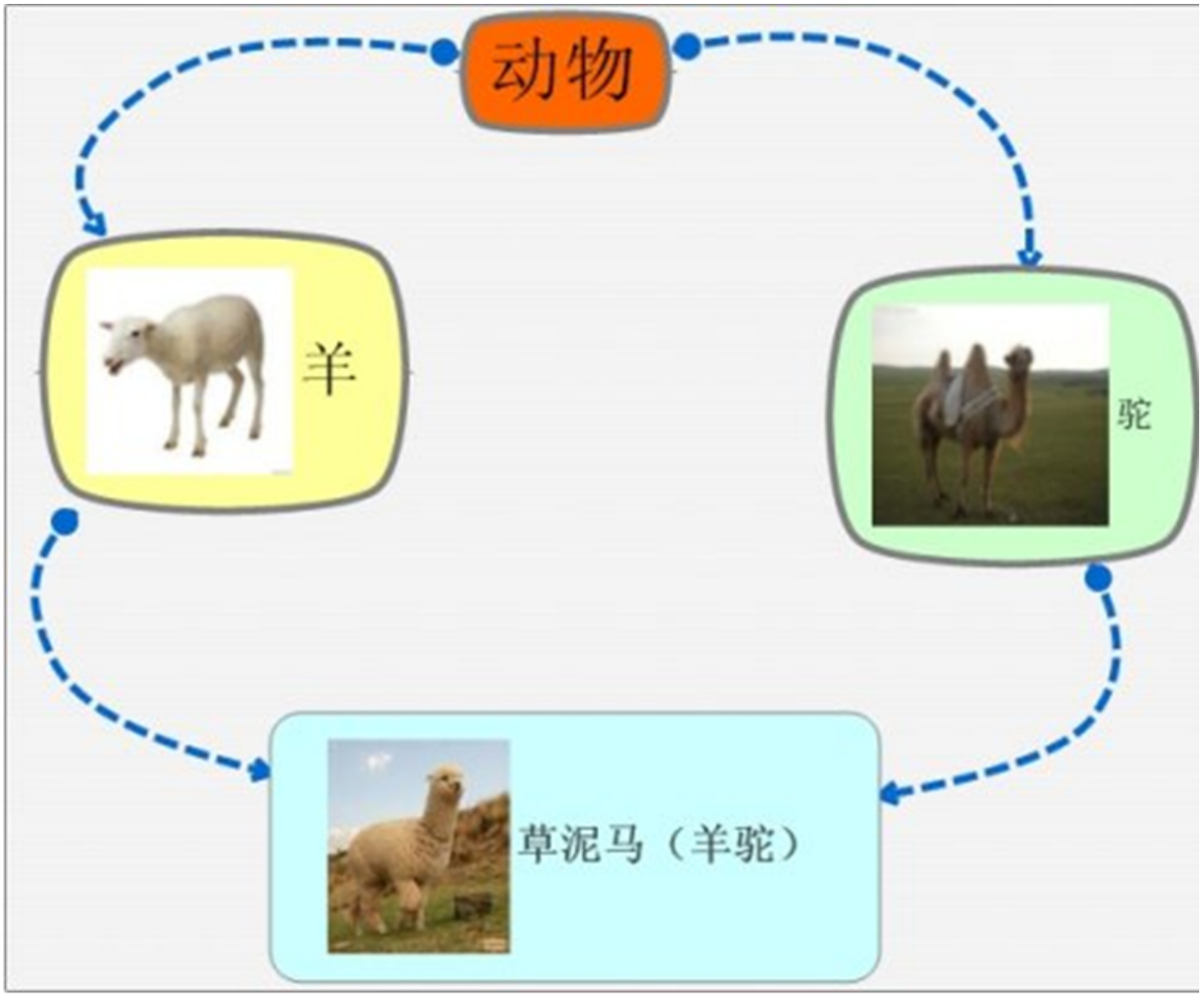
```
class Base1
{
public:
 int a;
 Base1(int a):a(a){}
};
class Base2
{
public:
  int a;
  Base2(int a):a(a){}
 };


 class Son:public Base1, public Base2
 {
 public:
  int a;
  Son(int a, int b,int c):Base1(a),Base2(b),a(c){}
 };

 void test01()
 {
  Son ob(10,20,30);
  cout<<ob.a<<endl;//子类a
  cout<<ob. Base1::a<<endl;//Base1 a
  cout<<ob.Base2::a<<endl;//Base2 a
 }
```



知识点8【菱形继承】（了解）

菱形继承：有公共祖先的继承 叫菱形继承。

最底层的子类 数据 会包含多分（公共祖先的数据）




```
class Animal
{
public:
 int data;
};
class Sheep :public Animal{};
```

```
class Tuo :public Animal {};
class SheepTuo:public Sheep,public Tuo{};
int main()
{
 SheepTuo ob;
 memset(&ob, 0, sizeof(SheepTuo));

 //cout << ob.data << endl;//二义性
 cout << ob.Sheep::data << endl;
 cout << ob.Tuo::data << endl;
}
```

怎么才能只要公共祖先的一份数据呢?

知识点9【虚继承】（了解）

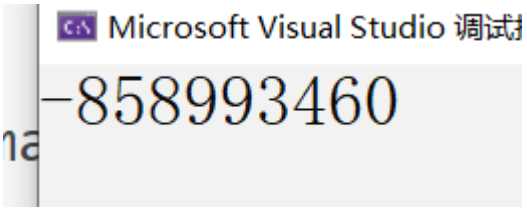
虚继承 解决 菱形继承中 多分公共祖先数据的问题

1、虚继承的概述

在继承方式 前加virtual修饰

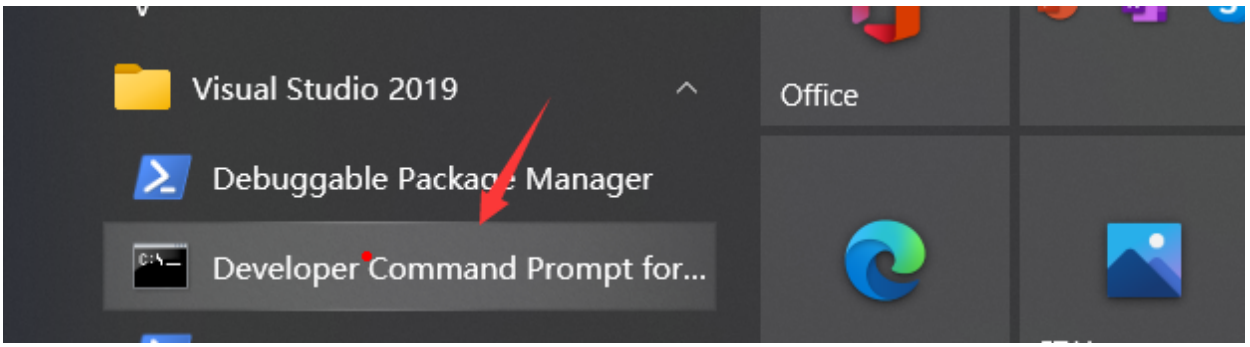
子类虚继承父类 子类只会保存一份公共数据。

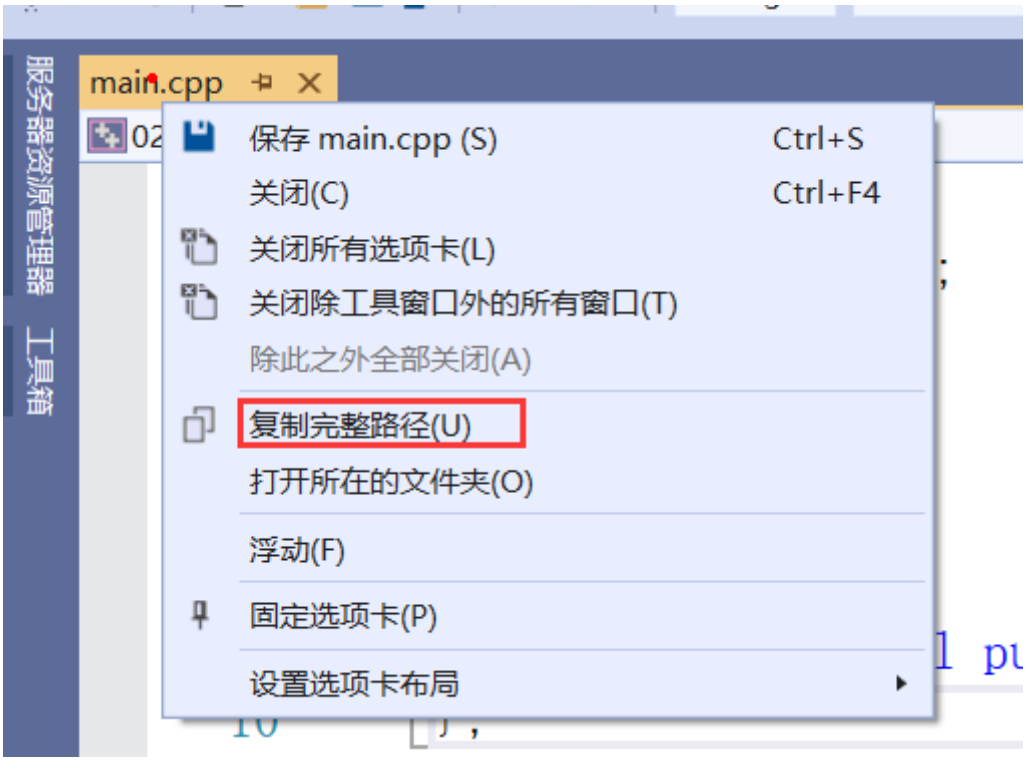
```
#include<iostream>
#include<string.h>
using namespace std;
class Animal
{
public:
 int data;
};
class Sheep :virtual public Animal{
};
class Tuo :virtual public Animal {
};
class SheepTuo:public Sheep,public Tuo{};
int main()
{
 SheepTuo ob;
 cout << ob.data << endl;
 return 0;
}
```



分析原理方法（vs有效）

2、分析虚继承的实现原理





导出类的布局：

cl /d1 reportSingleClassLayoutAnimal main.cpp

Animal布局：

```
class Animal     size(4):
        +---
 0      | data
        +---
```

Sheep布局:

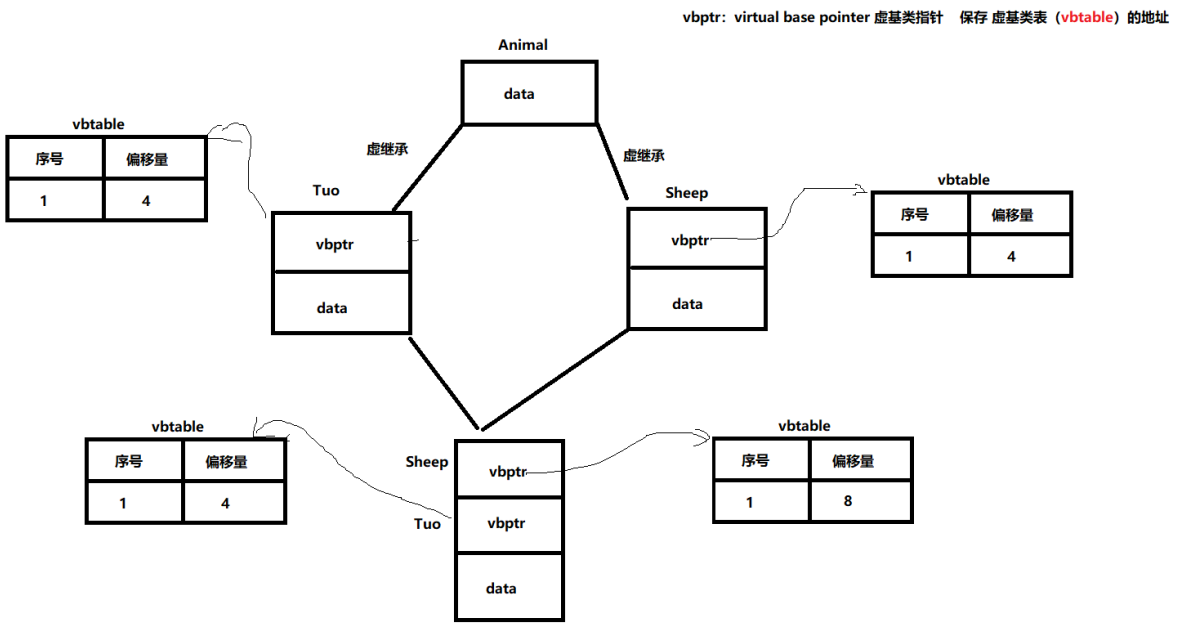
```
class Sheep     size(8):
        +---
 0      | {vbptr}
        +---
        +--- (virtual base Animal)
 4      | data
        +---

Sheep::$vbtable@:
 0      | 0
 1      | 4 (Sheepd(Sheep+0)Animal)
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
          Animal       4       0       4 0
```

Tuo布局:

```
class Tuo       size(8):
        +---
 0      | {vbptr}
        +---
        +--- (virtual base Animal)
 4      | data
        +---

Tuo::$vbtable@:
 0      | 0
 1      | 4 (Tuod(Tuo+0)Animal)
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
          Animal       4       0       4 0
```

SheepTuo布局:

```
class SheepTuo  size(12):
        +---
 0      | +--- (base class Sheep)
 0      | | {vbptr}
        | +---
 4      | +--- (base class Tuo)
 4      | | {vbptr}
        | +---
        +---
        +--- (virtual base Animal)
 8      | data
        +---

SheepTuo::$vbtable@Sheep@:
 0      | 0
 1      | 8 (SheepTuod(Sheep+0)Animal)

SheepTuo::$vbtable@Tuo@:
 0      | 0
 1      | 4 (SheepTuod(Tuo+0)Animal)
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
           Animal       8       0       4 0
```



虚继承 会在子类中产生 虚基类指针（vbptr） 指向 虚基类表(vbtable), 虚基类表纪录的是通过该指针访问公共祖先的数据的偏移量。

注意:

虚继承只能解决具备公共祖先的多继承所带来的二义性问题，不能解决没有公共祖先的多继承的.

工程开发中真正意义上的多继承是几乎不被使用，因为多重继承带来的代码复杂性远多于其带来的便利，多重继承对代码维护性上的影响是灾难性的，在设计方法上，任何多继承都可以用单继承代替。