

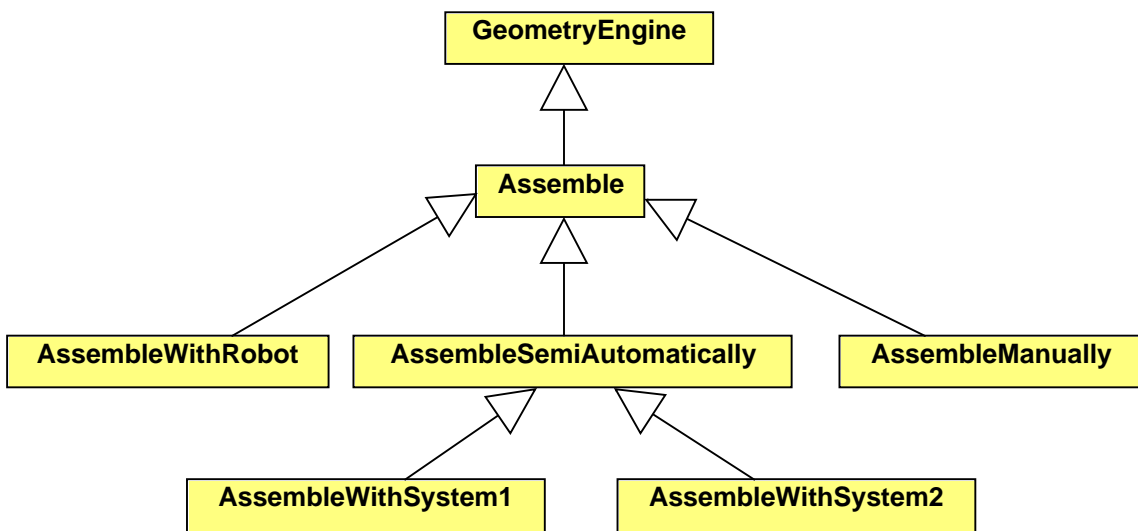
OO Design with Multiple Inheritance

Questions:

1. What is one single issue in C++ that you are allowed to become emotional about?

2. State a basic tenet of OO programming?

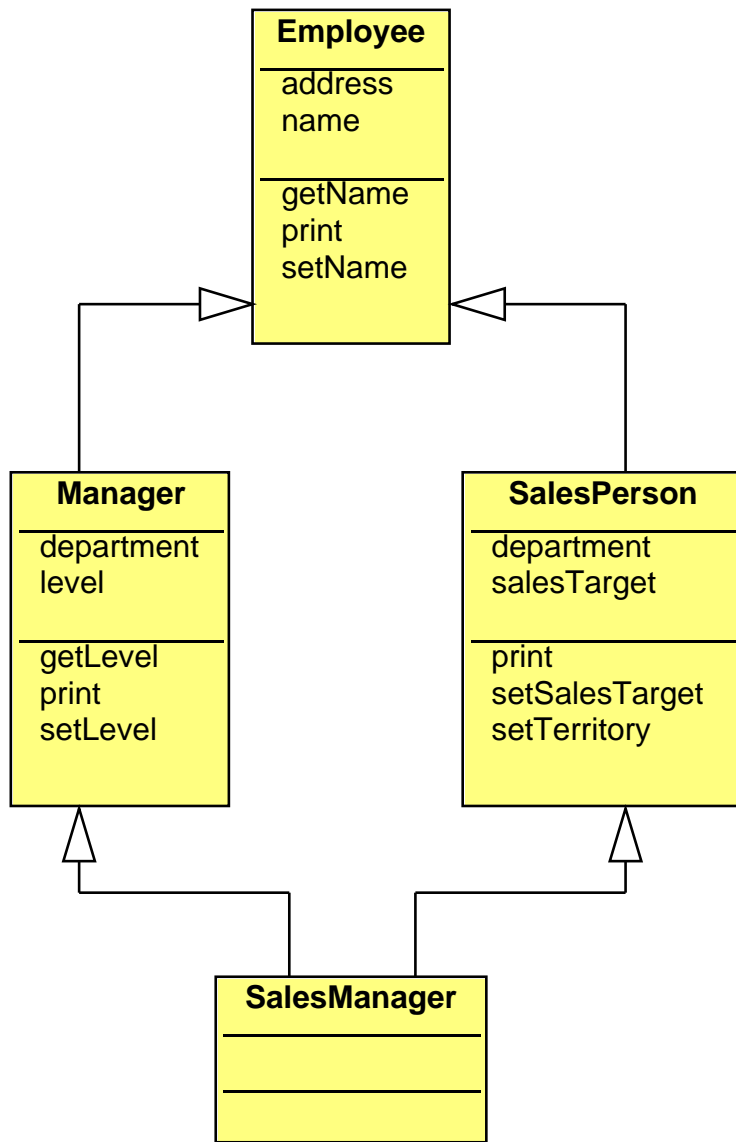
3. If the class Assemble represents at a purely abstract level the process of assembling together two mechanical widgets and if the class GeometryEngine contains the code for actually figuring out motion trajectories for the assembly, what's wrong with the following class hierarchy?



Issues that Arise with Repeated Inheritance

The most complicating issues that arise with MI have to do with what is known as *repeated inheritance*.

Repeated inheritance takes place when a derived class inherits the same members of some base class through two different paths in a class hierarchy.



The classes **Manager** and **SalesPerson** are both derived from the base class **Employee**.

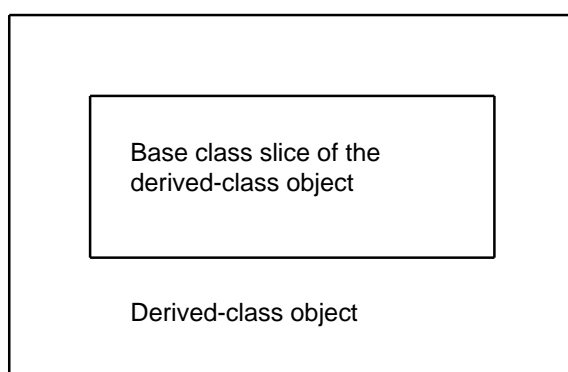
And the class **SalesManager** is derived from both **Manager** and **SalesPerson**.

We have intentionally left unspecified the data members and the member functions in the final derived-class **SalesManager**, as the following discussion bears directly on their specification.

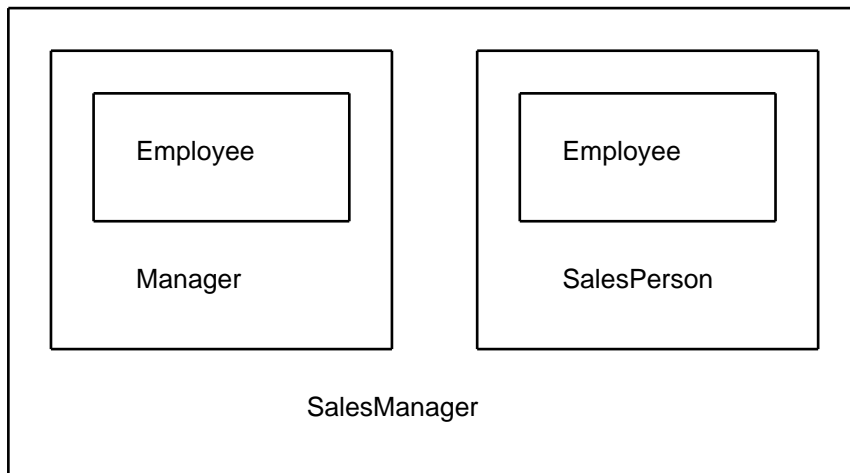
When a derived class can inherit members through multiple paths and when the different paths have an upstream class in common, the following issues become immediately relevant.

1. *The Problem of Duplicate Construction of Common-Base Subobject:*

Recall from earlier discussion that when the constructor of a derived class is invoked, the derived-class object so constructed has built inside it a base-class subobject.



That implies that, unless precautions are taken, when a constructor for **SalesManager** is invoked, we would end up with two different versions of the **Employee** slice in the constructed object.



How do we prevent the formation of duplicate common-base subobjects in a derived-class object?

2. *The Name-Conflict Problem for Member Functions:*

Suppose two member functions of the same signature but different implementations are inherited by the **SalesManager** class from the two different inheritance paths shown.

If these member functions are not overridden in the **SalesManager** class, we can end up with an ill-formed program.

This could, for example, be the case with the **print()** member function that is listed originally as a member of the class **Employee**.

Let's say this function is modified by each class between **Employee** and **SalesManager**.

If for some reason, this modified function is not overridden in **SalesManager** but yet invoked on an object of type **SalesManager**, you would have a compile-time ambiguity.

3. *The Name-Conflict Problem for Data Members:*

The class **SalesManager** inherits two different data members with the same name — **department** — one each from the two superclasses **Manager** and **SalesPerson**.

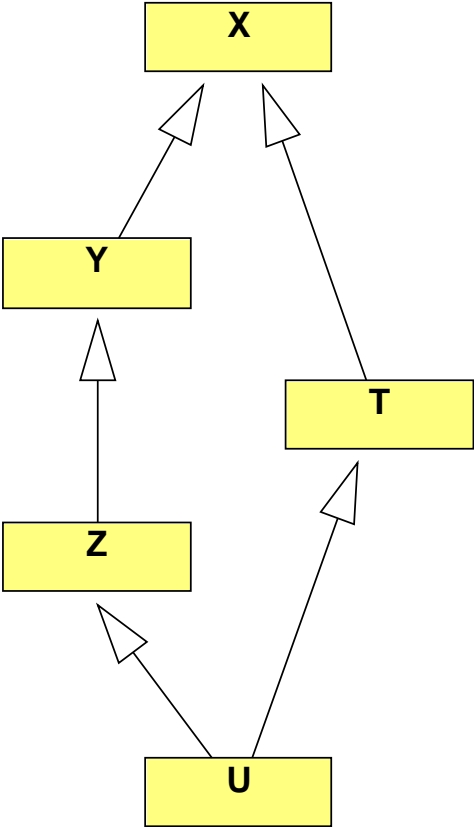
These two data members, although possessing the same name, possess different meanings for the derived class **SalesManager** because a **SalesManager** could conceivably have two different ‘department’ attributes associated with him or her.

Such an individual could belong to a particular department of the corporation and, at the same time, be in charge of a particular unit of the sales organization which could also be referred to as a department.

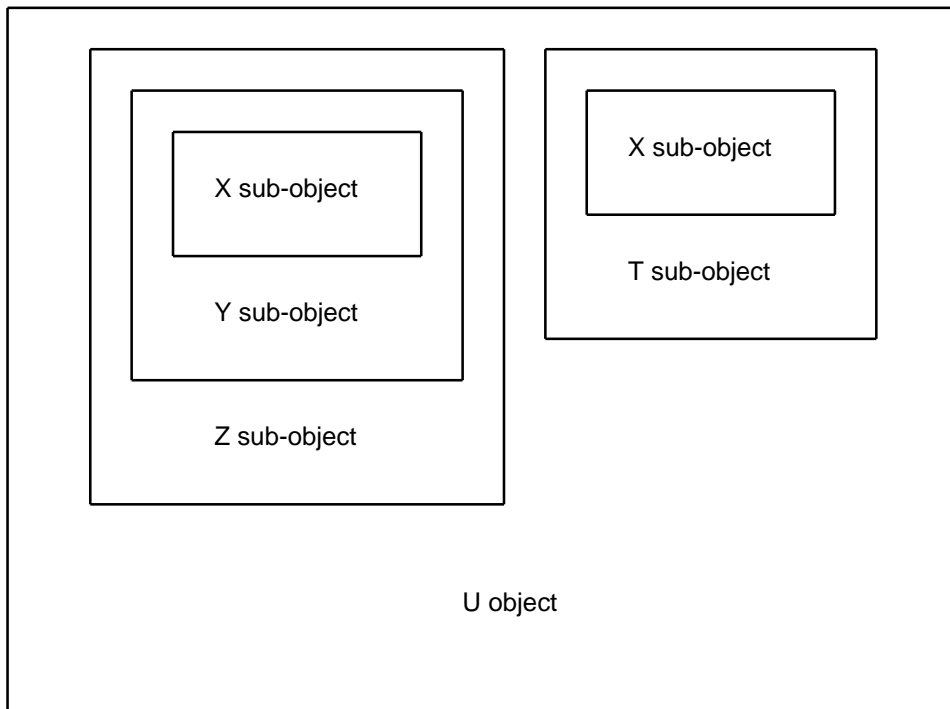
So how does one make sure that despite the same name — department — each data member gets the correct value when we construct an object of type **SalesManager**?

Virtual Bases for Multiple Inheritance

By declaring a common base to be virtual, we eliminate the problem of duplicate construction of sub-objects of the common base type.

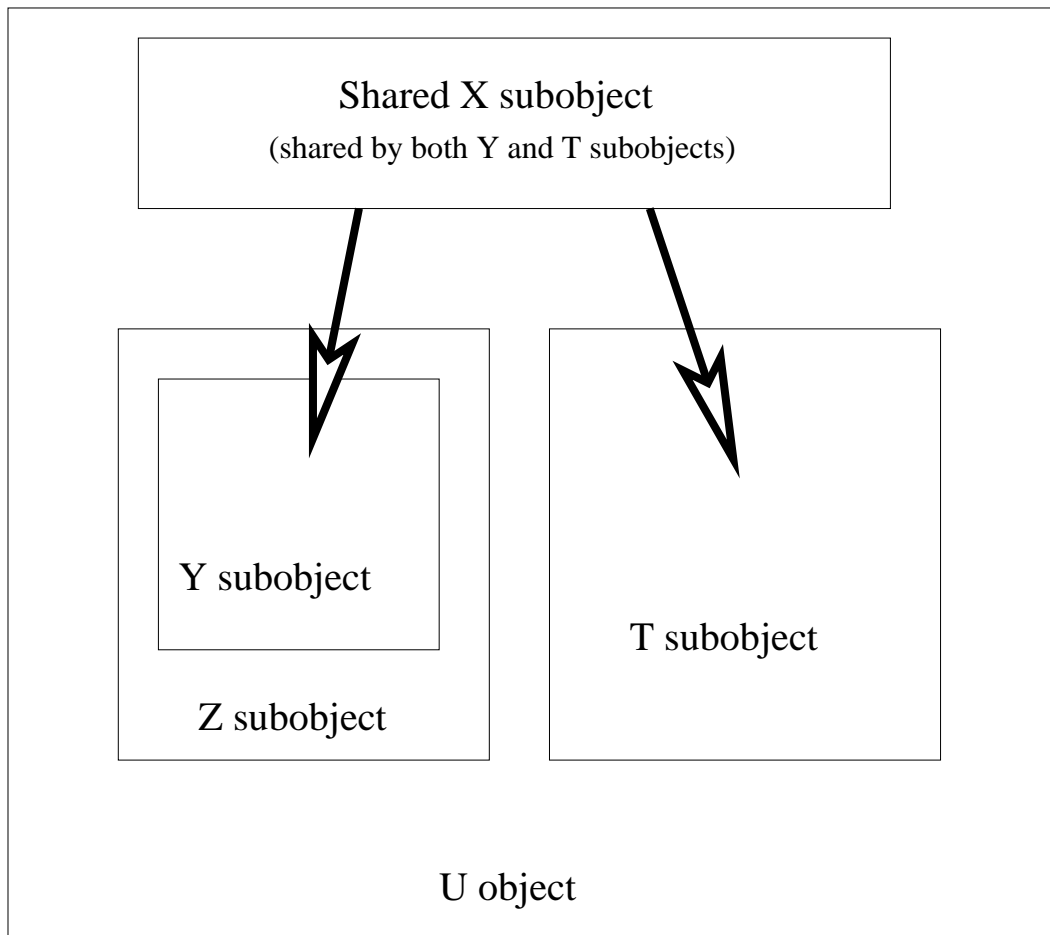


```
class Y : public X { ..... }  
class T : public X { ..... }  
class Z : public Y { ..... }  
class U : public Z, public T { ..... }
```



```
class Y : virtual public X {  
  //....  
};
```

```
class T : virtual public X {  
  //....  
};
```



While declaring **X** as a virtual base of **Y** and **T** does result in the sharing of the **X** sub-object, it requires that the constructor for **X** be now invoked explicitly in all the descendents of **X**.

Ordinarily, a derived class constructor cannot invoke the constructor of its "indirect" base classes. (A base class is an indirect base if it is not an immediate base class.)

```

#include <iostream>

class X {
    int x;
public:
    X( int xx ) : x(xx) {}
    virtual void print()
    {
        cout << "printing value of x of X sub-object: " << x << endl;
    }
};

class Y : virtual public X {
    int y;
public:
    Y( int xx, int yy ) : X( xx ), y( yy ) {}
    void print()
    {
        X::print();
        cout << "printing value of y of Y sub-object: " << y << endl;
    }
};

class T : virtual public X {
    int t;
public:
    T( int xx, int tt ) : X( xx ), t( tt ) {}
    void print()
    {
        X::print();
        cout << "printing value of t of T sub-object: " << t << endl;
    }
};

```

```

class Z : public Y {
    int z;
public:
    Z( int xx, int yy, int zz ) : Y( xx, yy ), X(xx), z( zz ) {}
    void print()
    {
        Y::print();
        cout << "printing value of z of Z sub-object: " << z << endl;
    }
};

```

```

class U : public Z, public T {
    int u;
public:
    U ( int xx, int yy, int zz, int tt, int uu )
        : Z( xx, yy, zz ), T( xx, tt ), X( xx ), u( uu ) {}
    void print()
    {
        Z::print();
        T::print();
        cout << "printing value of u of U sub-object: " << u << endl;
    }
};

```

```

int main()
{
    cout << "X object coming up: " << endl;
    X xobj( 1 );
    xobj.print();
}

```

```
cout << endl;

cout << "Y object coming up: " << endl;
Y yobj( 11, 12 );
yobj.print();
cout << endl;

cout << "Z object coming up: " << endl;
Z zobj( 110, 120, 130 );
zobj.print();
cout << endl;

cout << "T object coming up: " << endl;
T tobj( 21, 22 );
tobj.print();
cout << endl;

cout << "U object coming up: " << endl;
U uobj(9100, 9200, 9300, 9400, 9500 ); // (A)
uobj.print();
cout << endl;
}
```

X object coming up:
printing value of x of X sub-object: 1

Y object coming up:
printing value of x of X sub-object: 11
printing value of y of Y sub-object: 12

Z object coming up:
printing value of x of X sub-object: 110
printing value of y of Y sub-object: 120
printing value of z of Z sub-object: 130

T object coming up:
printing value of x of X sub-object: 21
printing value of t of T sub-object: 22

U object coming up:
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500

```
class U : public Z, public T {
    int u;
public:
    U ( int xx, int yy, int zz, int tt, int uu )
        : W( 4444, yy, zz ), T( 5555, tt ), X( 6666 ), u( uu ) {}
    void print()
    {
        W::print();
        T::print();
        cout << "printing value of u of U sub-object: " << u << endl;
    }
};
```

X object coming up:

printing value of x of X sub-object: 1

Y object coming up:

printing value of x of X sub-object: 11

printing value of y of Y sub-object: 12

Z object coming up:

printing value of x of X sub-object: 110

printing value of y of Y sub-object: 120

printing value of z of Z sub-object: 130

T object coming up:

printing value of x of X sub-object: 21

printing value of t of T sub-object: 22

U object coming up:

printing value of x of X sub-object: 6666 // (a)

printing value of y of Y sub-object: 9200 // (b)

printing value of z of Z sub-object: 9300 // (c)

printing value of x of X sub-object: 6666 // (d)

printing value of t of T sub-object: 9400 // (e)

printing value of u of U sub-object: 9500 // (f)

Virtual Bases and Copy Constructors

Since the presence of a virtual base caused the downstream constructor definitions to be modified, an obvious next question is: what about the implementation of the downstream copy constructors?

Yes, the copy constructors must now also make a direct invocation of **X**'s copy constructor.

```

#include <iostream>

class X {
    int x;
public:
    X( int xx ) : x(xx) {}
    X( const X& other ) : x( other.x ) {}          // copy constructor
    virtual void print()
    {
        cout << "printing value of x of X sub-object: " << x << endl;
    }
};

class Y : virtual public X {
    int y;
public:
    Y( int xx, int yy ) : X( xx ), y( yy ) {}
    Y( const Y& other ) : X( other ), y( other.y ) {} // copy const
    void print()
    {
        X::print();
        cout << "printing value of y of Y sub-object: " << y << endl;
    }
};

```

```

class T : virtual public X {
    int t;
public:
    T( int xx, int tt ) : X( xx ), t( tt ) {}
    T( const T& other ) : X( other ), t( other.t ) {} // copy const
    void print()
    {
        X::print();
        cout << "printing value of t of T sub-object: " << t << endl;
    }
};

```

```

class Z : public Y {
    int z;
public:
    Z( int xx, int yy, int zz ) : Y( xx, yy ), X(xx), z( zz ) {}
    Z( const Z& other ) // copy const
        : Y( other ), X( other ), z( other.z ) {}
    void print()
    {
        Y::print();
        cout << "printing value of z of Z sub-object: " << z << endl;
    }
};

```

```

class U : public Z, public T {
    int u;
public:
    U ( int xx, int yy, int zz, int tt, int uu )
        : Z( xx, yy, zz ), T( xx, tt ), X( xx ), u( uu ) {}
    U( const U& other ) // copy const
        : Z( other ), T( other ), X( other ), u( other.u ) {}
    void print()
    {
        Z::print();
        T::print();
        cout << "printing value of u of U sub-object: " << u << endl;
    }
};

```

```

int main()
{
    cout << "Z object coming up: " << endl;
    Z z_obj_1( 1110, 1120, 1130 );
    z_obj_1.print();
    cout << endl;

    cout << "Z's duplicate object coming up: " << endl;
    Z z_obj_2 = z_obj_1;
    z_obj_2.print();
    cout << endl;

    cout << "U object coming up: " << endl;
    U u_obj_1(9100, 9200, 9300, 9400, 9500 );
    u_obj_1.print();
    cout << endl;

    cout << "U's duplicate object coming up: " << endl;
    U u_obj_2 = u_obj_1;
    u_obj_2.print();
}

```

```
    cout << endl;  
}
```

This program produces the following output:

Z object coming up:

```
printing value of x of X sub-object: 1110
printing value of y of Y sub-object: 1120
printing value of z of Z sub-object: 1130
```

Z's duplicate object coming up:

```
printing value of x of X sub-object: 1110
printing value of y of Y sub-object: 1120
printing value of z of Z sub-object: 1130
```

U object coming up:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

U's duplicate object coming up:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
```

printing value of u of U sub-object: 9500

Virtual Bases and Assignment Operators

Given that a virtual base does impinge on the structure of the copy constructors for the derived classes, what about the assignment operator?

As it turns out, the syntax of the assignment operator function remains the same whether or not a base is virtual.

We will first show the program of the previous section with the assignment operator implementations added.

We will next show the output of the program of Item 1 when in the implementation code of the assignment operator of a class that is at the convergence point of two different inheritance paths, we intentionally suppress the assignments for one of the paths.

Finally, to make a graphic demonstration of the presence of duplicate sub-objects when a common base is not virtual, we will drop the “virtual” declaration of the common base in the program of Item 1 and make appropriate changes to the rest of the program to make it consistent with a non-virtual base.

```

#include <iostream>

class X {
    int x;
public:
    X( int xx ) : x(xx) {}
    X( const X& other ) : x( other.x ) {}

    X& operator=( const X& other )           // assignment op
    {
        if ( this == &other ) return *this;
        x = other.x;
        return *this;
    }

    virtual void print()
    {
        cout << "printing value of x of X sub-object: " << x << endl;
    }
};

class Y : virtual public X {
    int y;
public:
    Y( int xx, int yy ) : X( xx ), y( yy ) {}
    Y( const Y& other ) : X( other ), y( other.y ) {}

    Y& operator=( const Y& other )           // assignment op
    {
        if ( this == &other ) return *this;
        X::operator=( other );
        y = other.y;
        return *this;
    }

    void print()
    {
        X::print();
        cout << "printing value of y of Y sub-object: " << y << endl;
    }
};

```

```

class T : virtual public X {
    int t;
public:
    T( int xx, int tt ) : X( xx ), t( tt ) {}
    T( const T& other ) : X( other ), t( other.t ) {}

    T& operator=( const T& other )           // assignment op
    {
        if ( this == &other ) return *this;
        X::operator=( other );
        t = other.t;
        return *this;
    }

    void print()
    {
        X::print();
        cout << "printing value of t of T sub-object: " << t << endl;
    }
};

```

```

class Z : public Y {
    int z;
public:
    Z( int xx, int yy, int zz ) : Y( xx, yy ), X(xx), z( zz ) {}
    Z( const Z& other ) : Y( other ), X( other ), z( other.z ) {}

    Z& operator=( const Z& other )           // assignment op
    {
        if ( this == &other ) return *this;
        Y::operator=( other );
        z = other.z;
        return *this;
    }

    void print()
    {
        Y::print();
        cout << "printing value of z of Z sub-object: " << z << endl;
    }
};

```

```

class U : public Z, public T {
    int u;
public:
    U ( int xx, int yy, int zz, int tt, int uu )
        : Z( xx, yy, zz ), T( xx, tt ), X( xx ), u( uu ) {}
    U( const U& other )
        : Z( other ), T( other ), X( other ), u( other.u ) {}

```

```

U& operator=( const U& other )           // assignment op
{
    if ( this == &other ) return *this;
    Z::operator=( other );               // (A)
    T::operator=( other );               // (B)
    u = other.u;
    return *this;
}

void print()
{
    Z::print();
    T::print();
    cout << "printing value of u of U sub-object: " << u << endl;
}
};

```

```

int main()
{
    cout << "U object coming up: " << endl;
    U u_obj_1(9100, 9200, 9300, 9400, 9500 );
    u_obj_1.print();
    cout << endl;

    U u_obj_2(7100, 7200, 7300, 7400, 7500 );

    u_obj_2 = u_obj_1;

    cout << "U object after assignment: " << endl;
    u_obj_2.print();
}

```

U object coming up:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

The U object after assignment to another U object:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

U object coming up:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

The U object after assignment to another U object:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 7200
printing value of z of Z sub-object: 7300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

```
#include <iostream.h>
```

```
class X {
    int x;
public:
    X( int xx ) : x(xx) {}
    X( const X& other ) : x( other.x ) {}
    X& operator=( const X& other )
    {
        if ( this == &other ) return *this;
        x = other.x;
        return *this;
    }
    virtual void print()
    {
        cout << "printing value of x of X sub-object: " << x << endl;
    }
};
```

```
//class Y : virtual public X {
class Y : public X { // base non-virtual
    int y;
public:
    Y( int xx, int yy ) : X( xx ), y( yy ) {}
    Y( const Y& other ) : X( other ), y( other.y ) {}

    Y& operator=( const Y& other )
    {
        if ( this == &other ) return *this;
        X::operator=( other );
        y = other.y;
        return *this;
    }

    void print()
    {
        X::print();
        cout << "printing value of y of Y sub-object: " << y << endl;
    }
};
```

```

//class T : virtual public X {
class T : public X { // base non-virtual
    int t;
public:
    T( int xx, int tt ) : X( xx ), t( tt ) {}
    T( const T& other ) : X( other ), t( other.t ) {}

    T& operator=( const T& other )
    {
        if ( this == &other ) return *this;
        X::operator=( other );
        t = other.t;
        return *this;
    }

    void print()
    {
        X::print();
        cout << "printing value of t of T sub-object: " << t << endl;
    }
};

class Z : public Y {
    int z;
public:
    // Z( int xx, int yy, int zz ) : Y( xx, yy ), X(xx), z( zz ) {}
    Z( int xx, int yy, int zz ) : Y( xx, yy ), z( zz ) {}
    // Z( const Z& other ) : Y( other ), X( other ), z( other.z ) {}
    Z( const Z& other ) : Y( other ), z( other.z ) {}

    Z& operator=( const Z& other )
    {
        if ( this == &other ) return *this;
        Y::operator=( other );
        z = other.z;
        return *this;
    }

    void print()
    {
        Y::print();
        cout << "printing value of z of Z sub-object: " << z << endl;
    }
};

```

```

class U : public Z, public T {
    int u;
public:
    U ( int xx, int yy, int zz, int tt, int uu )
        //      : Z( xx, yy, zz ), T( xx, tt ), X( xx ), u( uu ) {}
        //      : Z( xx, yy, zz ), T( xx, tt ), u( uu ) {}
    U( const U& other )
        //      : Z( other ), T( other ), X( other ), u( other.u ) {}
        //      : Z( other ), T( other ), u( other.u ) {}
    U& operator=( const U& other )
    {
        if ( this == &other ) return *this;
        Z::operator=( other );           // (A)
        T::operator=( other );           // (B)
        u = other.u;
        return *this;
    }

    void print()
    {
        Z::print();
        T::print();
        cout << "printing value of u of U sub-object: " << u << endl;
    }
};

int main()
{
    cout << "U object coming up: " << endl;
    U u_obj_1(9100, 9200, 9300, 9400, 9500 );
    u_obj_1.print();
    cout << endl;

    U u_obj_2(7100, 7200, 7300, 7400, 7500 );

    u_obj_2 = u_obj_1;

    cout << "The U object after assignment to another U object: " << endl;
    u_obj_2.print();
    cout << endl;
}

```

U object coming up:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

The U object after assignment to another U object:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

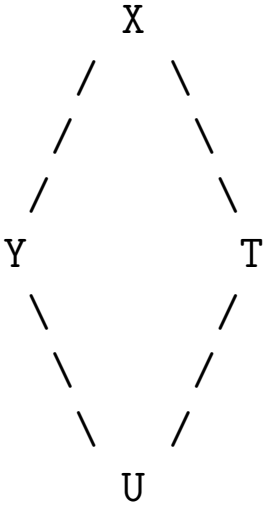
U object coming up:

```
printing value of x of X sub-object: 9100
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 9100
printing value of t of T sub-object: 9400
printing value of u of U sub-object: 9500
```

The U object after assignment to another U object:

```
printing value of x of X sub-object: 9100      // (C)
printing value of y of Y sub-object: 9200
printing value of z of Z sub-object: 9300
printing value of x of X sub-object: 7100      // (D)
printing value of t of T sub-object: 7400
printing value of u of U sub-object: 9500
```

Avoiding Name Conflicts for Methods



```

#include <iostream>

class X {
public:
    void foo() { cout << "inside foo as defined in X" << endl; }
};

class Y : virtual public X {
public:
    void foo() { cout << "inside foo as overridden by Y" << endl; }
};

class T : virtual public X {
public:
    void foo() { cout << "inside foo as overridden by T" << endl; } // (A)
};

class U : public Y, public T {
public:
    void foo() { cout << "inside foo as overridden by U" << endl; } // (B)
};

int main()
{
    cout << "U object coming up: " << endl;
    U u_obj;
    u_obj.foo();
}

```

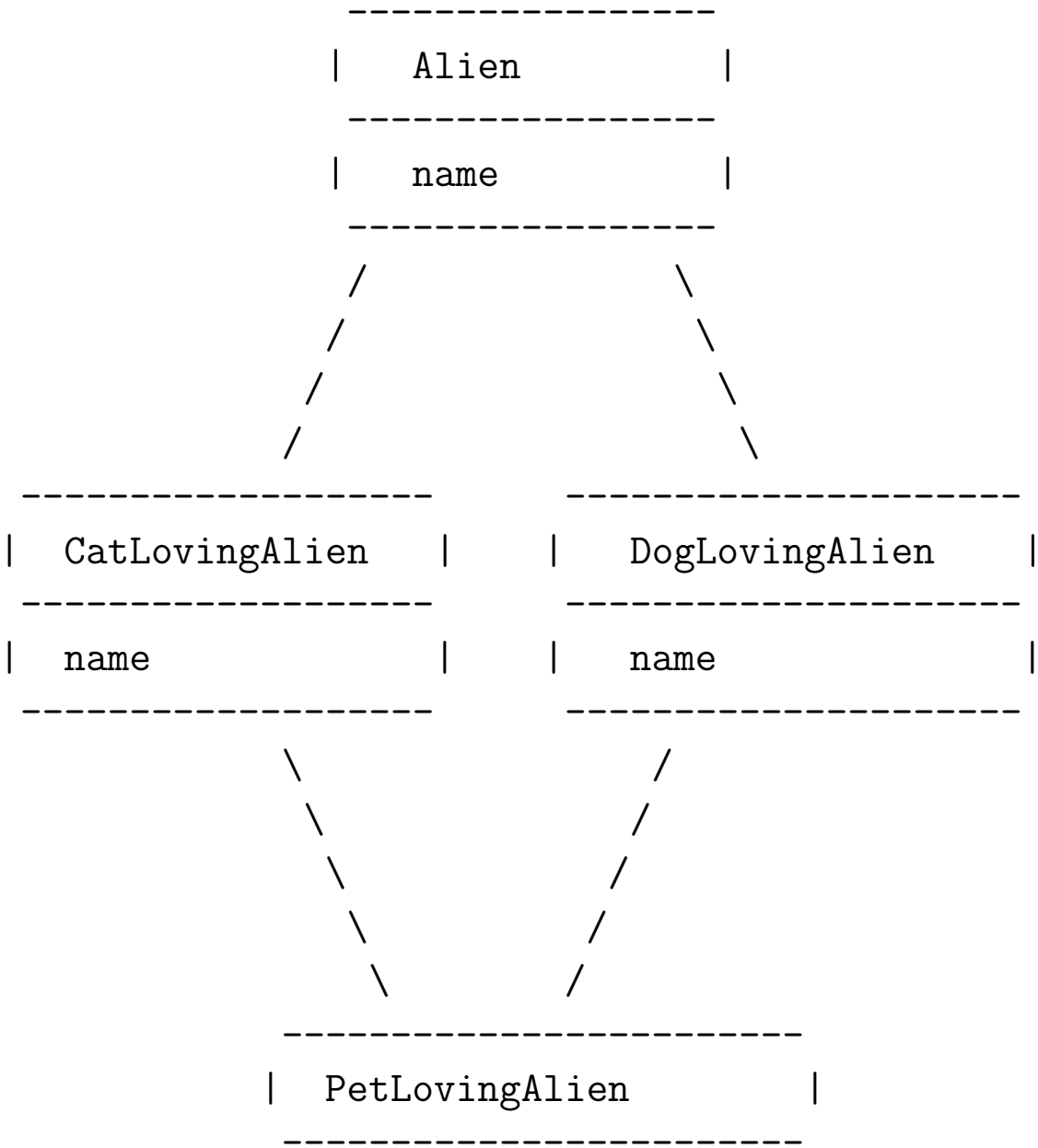
Even though `U` inherits two different version of `foo()` from the different inheritance paths, it does not cause any problems since `U` has its override definition for `foo()`.

`U` object coming up:

inside `foo` as overridden by `U`

When a function is inherited from two different inheritance paths, it is the more recently overridden definition of the function that is used in the common derived class (provided there is only one such overridden definition).

Dealing with Name Conflicts for Data Members



```
PetLovingAlien( string catName, string dogName, string ownerName  
    : CatLovingAlien( catName, ownerName ),  
    DogLovingAlien( dogName, ownerName ),  
    Alien( ownerName ) {}
```

```
#include <iostream>
#include <string>
```

```
class Alien {
protected:
    string name;
public:
    Alien( string nam ) : name( nam ) {}
};
```

```
class CatLovingAlien : virtual public Alien {
protected:
    string name;           // Cat's name
public:
    CatLovingAlien( string catName, string ownerName )
        : Alien( ownerName), name ( catName ) {}
};
```

```
class DogLovingAlien : virtual public Alien {
protected:
    string name;           // Dog's name
public:
    DogLovingAlien( string dogName, string ownerName )
        : Alien( ownerName ), name( dogName ) {}
};
```

```
class PetLovingAlien : public CatLovingAlien, public DogLovingAlien {
public:
    PetLovingAlien( string catName, string dogName, string ownerName )
        : CatLovingAlien( catName, ownerName ),
```

```

        DogLovingAlien( dogName, ownerName ),
        Alien( ownerName ) {}
void print()
{
    cout << CatLovingAlien::name << " "
         << DogLovingAlien::name << " "
         << Alien::name
         << endl;
}
};

int main()
{
    PetLovingAlien al( "Kitty", "Glutton", "Zaphod" );
    al.print();
}

```