类的多重继承：从多于一个直接基类进行继承。

非虚拟继承导致的二义性：

例：程序class\_test32

// 非虚拟继承导致的二义性

#include <iostream>

using namespace std;

class Furniture

{

public:

Furniture()

{

cout << "Furniture()" << endl;

}

void SetWeight(int i)

{

weight\_ = i;

}

int GetWeight()

{

return weight\_;

}

protected:

int weight\_;

};

class Bed : public Furniture

{

public:

Bed()

{

cout << "Bed()" << endl;

}

void Sleep()

{

cout << "Sleeping...\n";

}

protected:

int height\_;

};

class Sofa : public Furniture

{

public:

Sofa()

{

cout << "Sofa()" << endl;

}

void WatchTV()

{

cout << "WatchTV.\n";

}

protected:

int time\_;

};

class SleepSofa : public Bed, public Sofa

{

public:

SleepSofa()

{

}

void FoldOut()

{

cout << "fold out the sofa.\n";

}

};

int main(int argc, char\* argv[])

{

// 可以看出，调用了两次Furniture的构造函数

SleepSofa ss;

// ss.SetWeight(20); // error，对函数SetWeight的访问不明确

return 0;

}

在上例中，SleepSofa对象的内存布局如下：





通过虚拟继承来消除二义性：

在下例中，给之前的各个类的构造函数添加了参数：

例：程序class\_test31

// 多重继承和虚拟继承

#include <iostream>

using namespace std;

class Furniture

{

public:

Furniture(int w)

{

weight\_ = w;

cout << "Furniture()" << endl;

}

void SetWeight(int i)

{

weight\_ = i;

}

int GetWeight()

{

return weight\_;

}

protected:

int weight\_;

};

class Bed : virtual public Furniture // 通常情况下，将中间基类进行虚派生

{

public:

Bed(int h) : Furniture(20)

{

height\_ = h;

cout << "Bed()" << endl;

}

void sleep()

{

cout << "Sleeping...\n";

}

protected:

int height\_;

};

class Sofa : virtual public Furniture

{

public:

Sofa(int t) : Furniture(30)

{

time\_ = t;

cout << "Sofa()" << endl;

}

void WatchTV()

{

cout << "WatchTV.\n";

}

protected:

int time\_;

};

class SleepSofa : public Bed, public Sofa

{

public:

// 在SleepSofa的构造函数中，必须显示初始化基类Furniture

// 而不能只初始化SleepSofa的直接基类

SleepSofa() : Bed(50), Sofa(60), Furniture(80)

{

cout << "SleepSofa() " << "weight = " << weight\_ << endl;

cout << "height = " << height\_ << endl;

cout << "time = " << time\_ << endl;

}

void FoldOut()

{

cout << "fold out the sofa.\n";

}

};

int main(int argc, char\* argv[])

{

SleepSofa ss;

return 0;

}



可以看出，在SleepSofa的构造函数初始化列表中，必须调用顶层的基类Furniture的构造函数，原因是：Bed类Sofa类中共享了一个公共基类Furniture的对象副本，如果不在SleepSofa中显示调用Furniture的构造函数，则Bed和Sofa类都试图初始化这个Furniture的对象。

在SleepSofa的构造函数中调用Furniture的构造函数，在构造SleepSofa时，执行如下：

1. 构造Furniture部分；
2. 构造Bed部分，忽略Bed构造函数中，对Furniture的初始化；
3. 构造Sofa部分，忽略Sofa构造函数中，对Furniture的初始化；
4. 构造SleepSofa部分。

多重继承中的构造顺序

1. 任何虚拟基类的构造函数按照它们被继承的顺序构造；
2. 任何非虚拟基类的构造函数按照它们被继承的顺序构造；
3. 任何成员对象的构造函数按照它们声明的顺序调用；
4. 类自己的构造函数。

如下例所示：

例：程序class\_test33

// 多重继承构造顺序

#include <iostream>

using namespace std;

class OBJ

{

public:

OBJ()

{

cout << "OBJ\n";

}

};

class Base1

{

public:

Base1()

{

cout << "Base1\n";

}

};

class Base2 : virtual public Base1

{

public:

Base2()

{

cout << "Base2\n";

}

};

class Base3

{

public:

Base3()

{

cout << "Base3\n";

}

};

class Base4

{

public:

Base4()

{

cout << "Base4\n";

}

};

class Derived : public Base2 , public Base3, virtual public Base4

{

public:

Derived()

{

cout << "Derived\n";

}

protected:

OBJ obj;

};

int main(int argc, char\* argv[])

{

Derived d;

}

输出为：

Base1

Base4

Base2

Base3

OBJ

Derived

原因如下：

1. 首先调用Base1的构造函数，原因是Derived类继承了Base2类，而Base2类虚拟继承了Base1类，则Derived类虚拟继承了Base1类，按照多重继承的构造顺序，Base1的继承顺序在Base4之前。
2. 然后调用Base4的构造函数，原因是Base4是虚拟继承。
3. 再调用Base2的构造函数，原因是Base2的继承顺序在Base3之前。
4. 再调用Base3的构造函数。
5. 调用OBJ的构造函数，其为Derived的成员。
6. 最后调用Derived的构造函数。