第八章 继承与派生

模块8.2：多重继承的概念与实现

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

目录

- 多重继承
  - 多重继承的声明及使用
  - 多重继承的派生类
  - 多重继承引起的二义性问题

1.1 多重继承的声明及使用

- 声明形式

```
class 派生类名: private/protected/public 基类名1, ... ,
                                private/protected/public 基类名n {
        private:
            私有成员;
        protected:
            保护成员;
        public:
            公有成员;
};
```

1.1 多重继承的声明及使用

多重继承的使用

- 每个基类的基类存取限定符允许不同
- 每个基类被继承后的可访问性由其基类存取限定符确定
- 派生类继承所有基类的数据成员
- 派生类继承所有基类除构造函数和析构函数外的所有成员函数
- 派生类对象所占的空间 = 所有基类的数据成员之和 + 派生类的数据成员之和

1.1 多重继承的声明及使用

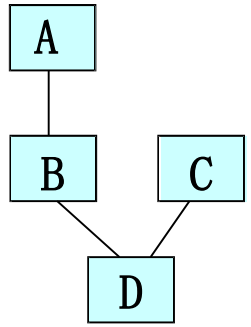
- 多重继承允许多层派生

```
class A {
  private:
    int a1;
  protected:
    int a2;
  public:
    int a3;
};
```

```
class B:public A {
  private:
    int b1;
  protected:
    int b2;
  public:
    int b3;
};
```

```
class C {
  private:
    int c1;
  protected:
    int c2;
  public:
    int c3;
};
```

```
class D:private B, public C {
  ...
};
```



1.2 多重继承的派生类

- 构造函数的形式

  派生类名(参数)：基类名1(参数),...， 基类名n(参数)

  {

  函数体(一般是对派生类新增成员的初始化)

  }

- 构造函数的使用：
  - 基类名出现的顺序无限制
  - 若调用基类的无参构造函数，则该基类可不出现在初始化参数表中
  - 若是多层派生，只出现直接基类
  - 若派生中含有基类的实例对象(子对象)，则子对象也可以出现在初始化参数表中

- 构造函数的使用：
  - 基类名出现的顺序无限制

```
class A {
  public:
    A(int x) { … }
}
class B {
  public:
    B(int y) { … }
}
```

```
class C:public A, private B{
  public:
    C(int a, int b, int c):A(a), B(b) {
                              B(b), A(a)
        …
    }
};
```

- 构造函数的使用：

  ➢ 若调用基类的无参构造函数，则该基类可不出现在初始化参数表中

```
class A {
  public:
    A() { … }
}
class B {
  public:
    B(int y) { … }
}
```

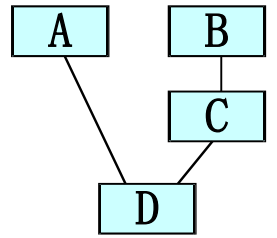
```
class C:public A, private B {
  public:
    C(int a, int b, int c):B(b) { … }
};
```

激活A的无参构造

- 构造函数的使用：
  - 若是多层派生，只出现直接基类

```
class A {
  public:
    A(int x) { … }
}
class B {
  public:
    B(int y) { … }
}
```

```
class C:public B {
  public:
     C(int z):B(z) { … };
}
class D:public A, private C {
  public:
     D(int a, int c, int d):A(a),C(c) { … }
};
```



B不需要出现

- 构造函数的使用：
  - 若派生中含有基类的实例对象(子对象)，则子对象也可以出现在初始化参数表中

```
class A {
  public:
    A(int x) { … }
}
class B {
  public:
    B(int y) { … }
}
```

```
class C:public A, private B {
  public:
    B bb;
    C(int a, int b, int b1):A(a), B(b), bb(b1){
    …
  }
};
```

1.2 多重继承的派生类

- 构造函数的调用顺序：
  - 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数
- 析构函数的调用顺序：
  - 先派生类的析构函数，再按声明顺序的反序依次调用基类的析构函数

//例1：

```
#include <iostream>
using namespace std;
class A {
   public:
      A() {cout << "A()" << endl;}
};
class B {
   public:
      B() {cout << "B()" << endl;}
};
class C {
   public:
      C() {cout << "C()" << endl;}
};
```

```
class D:public A, protected B, private C{
    public:
       D() {cout << "D()" << endl;}
};    //D():A(),B(),C() {…}
      //D():B(),A(),C() {…}
      //D():C(),B(),A() {…}
      …
int main(){
    D d1;
}
```

D对象的内存映像：

A的数据成员
B的数据成员
C的数据成员
D的数据成员

//例2：

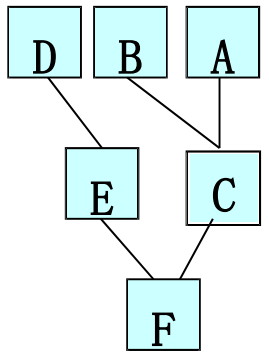
```
#include <iostream>
using namespace std;
class A {
  public:
    int a;
    A(){ a=1; }
};
class B {
  public:
    int b;
    B(){ b=2;}
};
class C {
  public:
    int c;
    C(){ c=3; }
};
class D:public A,
          protected B,
            private C {
 public:
    int d;
    D(){ d=4; }
};
int main()
{
    D d1;
    cout << sizeof(d1) << endl;
    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```

```
16
1
2
3
4
```

//例3：

```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B {
  public:
    B() {cout << "B()" << endl;}
};
class C:public B, public A {
  public:
    C() {cout << "C()" << endl;}
};
class D {
  public:
    D() {cout << "D()" << endl;}
};
class E:public D {
  public:
    E() {cout << "E()" << endl;}
};
class F:public E, public C {
  public:
    F() {cout << "F()" << endl;}
};
int main(){
    F f1;
}
```

```
D()
E()
B()
A()
C()
F()
```



F对象的内存映像：

1.3 多重继承引起的二义性问题

- 成员同名
  - 以不产生二义性为基本准则

某一个基类与派生类的成员同名：

按单继承的方式进行处理(支配规则：派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

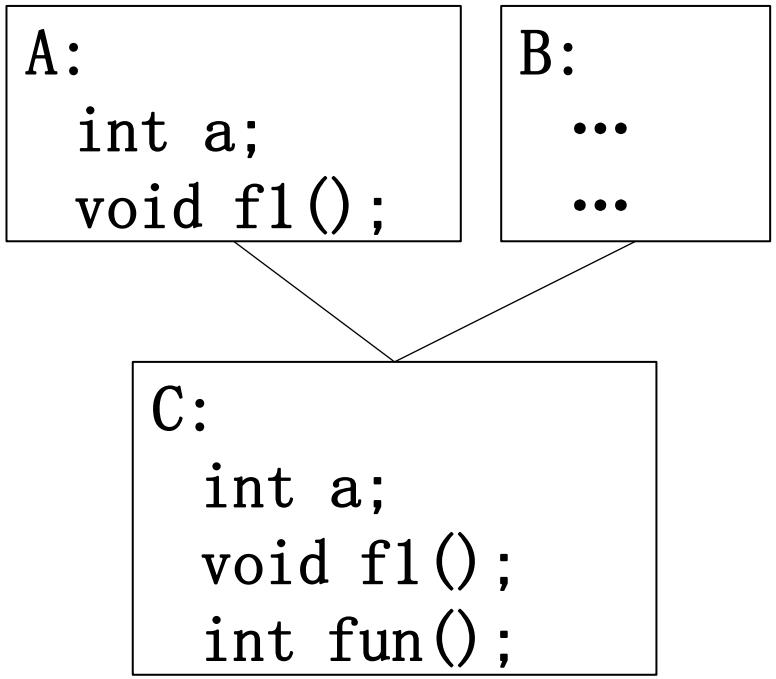
派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

- 某一个基类与派生类的成员同名：按单继承的方式进行处理

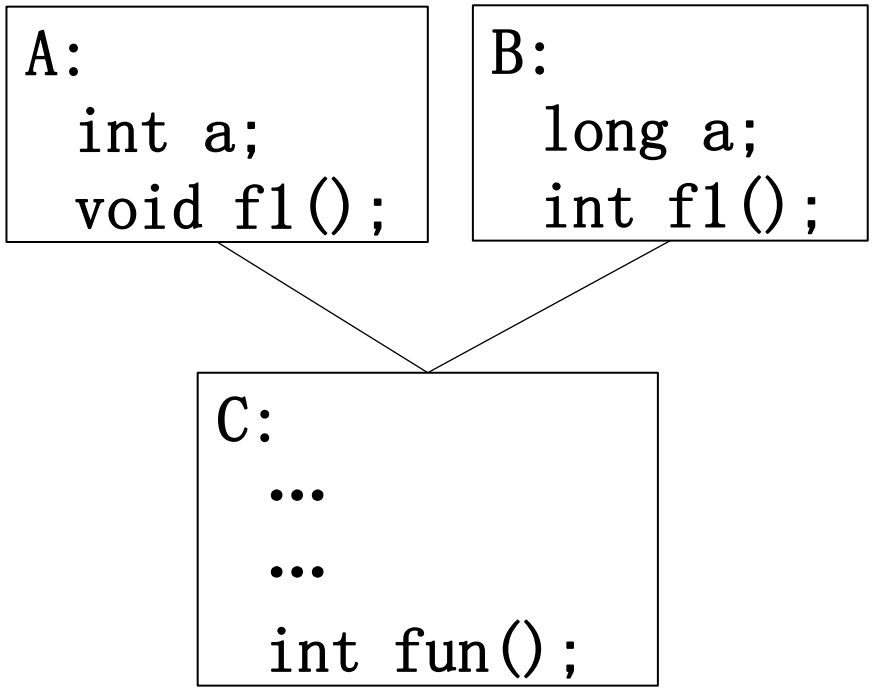
  （支配规则:派生类直接访问，基类加作用域符）




```
int C::fun()
{
    a=10;
    f1();
    A::a=15;
    A::f1();
}
```

```
int main()
{
    C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
}
```

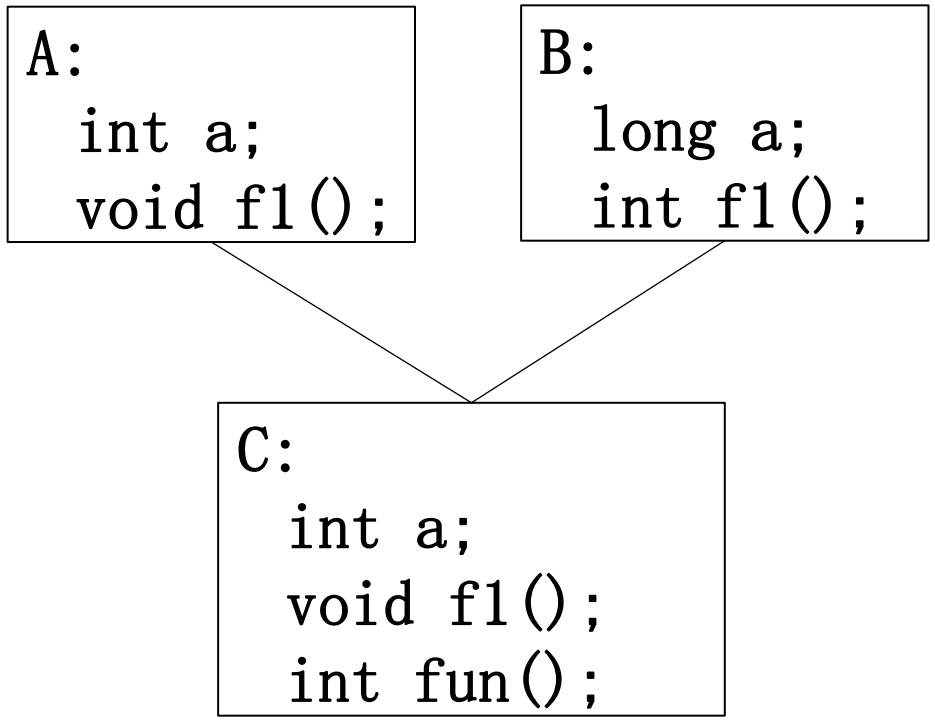
- 两个以上的基类中的成员同名：分别加不同的基类作用域符区分




```
int C::fun()
{
    A::a=10;
    A::f1();
    B::a=15;
    B::f1();
}
```

```
int main()
{
    C c1;
    c1.A::a=10;
    c1.A::f1();
    c1.B::a=15;
    c1.B::f1();
}
```

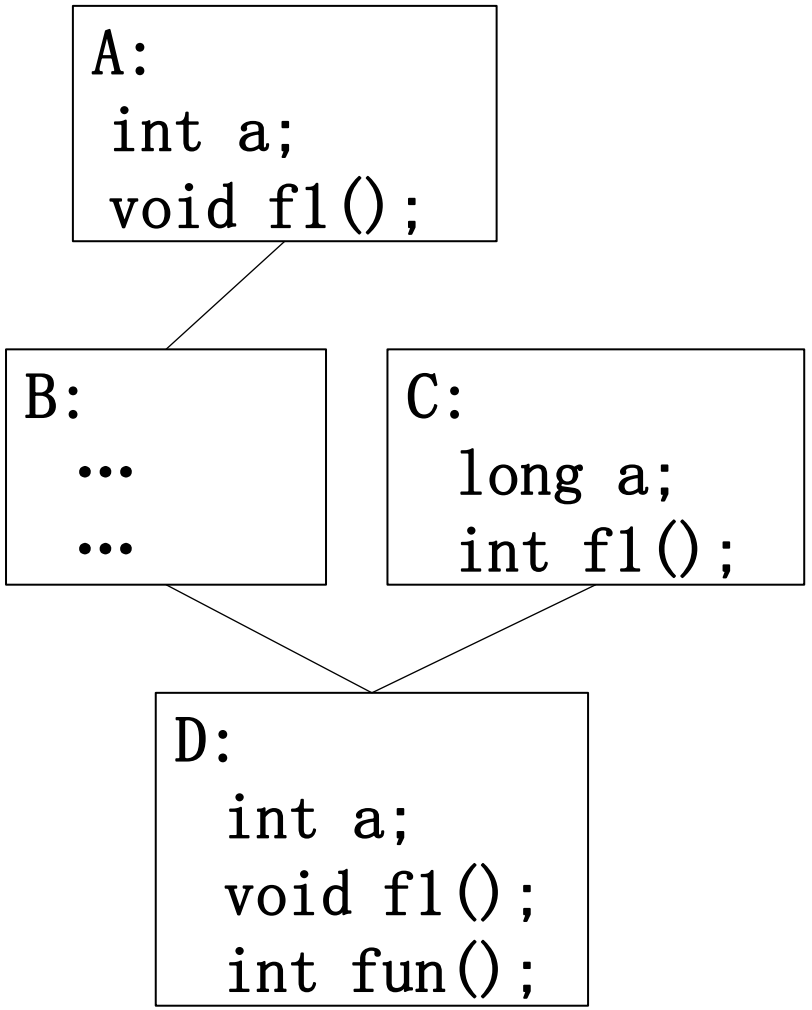
- 两个以上的基类与派生类的成员同名：派生类直接访问，不同基类加作用域符区分




```
int C::fun()
{
    a=10;
    f1();
    A::a=15;
    A::f1();
    B::a=20;
    B::f1();
}
```

```
int main()
{   C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
    c1.B::a=20;
    c1.B::f1();
}
```

- 通过直接/间接方式继承同一个基类两个导致的同名：通过可区分的类的作用域符来进行区分

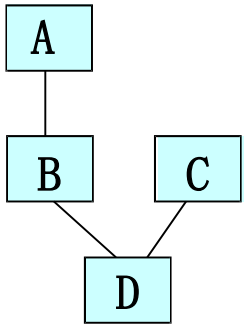



```
int D::fun()
{
   a=10;
   f1();
   A::a=15;可以B
   A::f1();可以B
   C::a=20;
   C::f1();
}
```

```
int main()
{  D d1;
   d1.a=10;
   d1.f1();
   d1.A::a=15;可以B
   d1.A::f1();可以B
   d1.C::a=20;
   d1.C::f1();
}
```

```
#include <iostream>
using namespace std;
class A {
  public:
    int a;
    void f1() { return; }
};
class B:public A {
  public:
    int b;
};
class C {
  public:
    long a;
    int f1() { return 0; }
};
class D:public B, public C {
  public:
    int a;
    char f1() { return 'A'; }
    void fun();
};
```

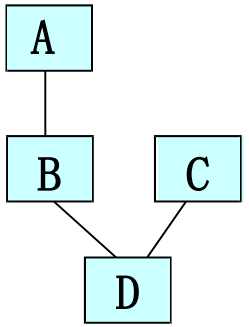
```
void D::fun(){
      a = 10;
      f1();
      A::a = 15; // B::a = 15;
      A::f1();   // B::f1();
      C::a = 20;
      C::f1();
      int *p = (int *)this;
      cout << sizeof(*this) <<endl;
      cout << *p << endl;
      cout << *(p+1) << endl;
      cout << *(p+2) << endl;
      cout << *(p+3) << endl;
}
int main(){
      D d1;
      d1.fun();
      return 0;
}
```




```
16
15
-858993460
20
10
```

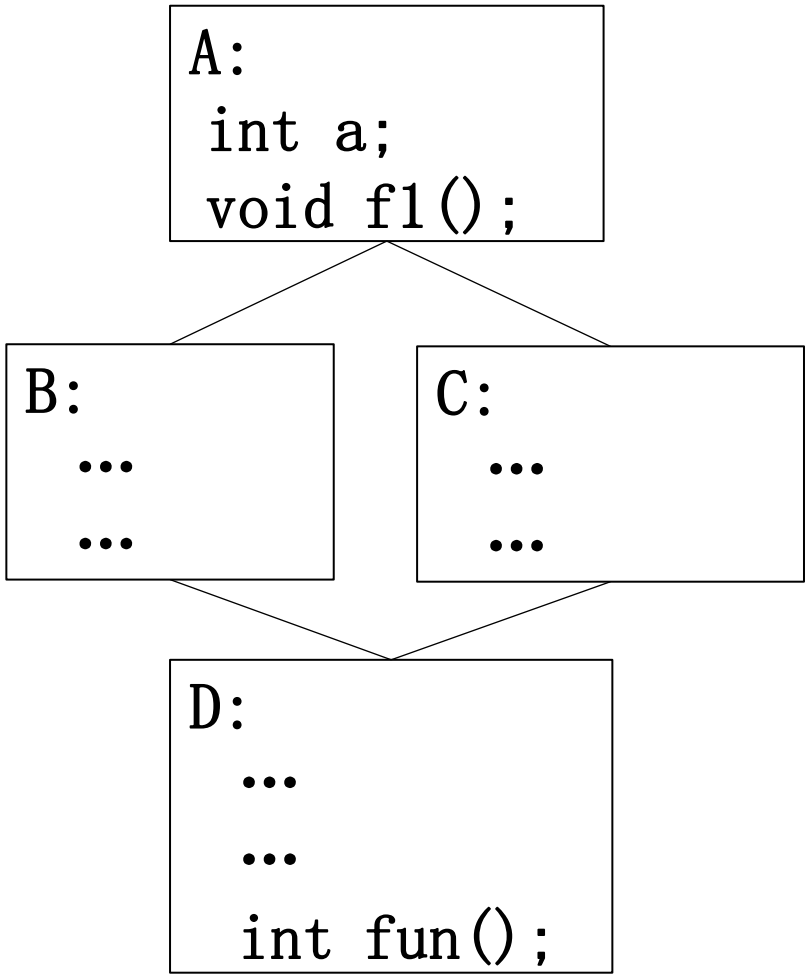
```
#include <iostream>
using namespace std;
class A {
  public:
    int a;
    void f1() { return; }
};
class B:public A {
  public:
    int b;
};
class C {
  public:
    long a;
    int f1() { return 0; }
};
class D:public B, public C {
  public:
    int a;
    char f1() { return 'A'; }
    void fun();
};
```

```
void D::fun(){
    return;
}
int main(){
    D d1;
    d1.a = 10;
    d1.f1();
    d1.A::a = 15;    // B::a = 15;
    d1.A::f1();      // B::f1();
    d1.C::a = 20;
    d1.C::f1();
    int *p = (int *)&d1;
    cout << sizeof(d1) <<endl;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```




```
16
15
-858993460
20
10
```

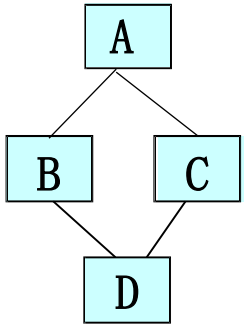
- 通过直接/间接方式继承同一个基类两个导致的同名：
  - 问题1：有几份A?



D对象的内存映象(两份A)

<table>
<tr><td>A的数据成员</td><td rowspan="2">B</td></tr>
<tr><td>B的数据成员</td></tr>
<tr><td>A的数据成员</td><td rowspan="2">C</td></tr>
<tr><td>C的数据成员</td></tr>
<tr><td>D的数据成员</td><td>D</td></tr>
</table>

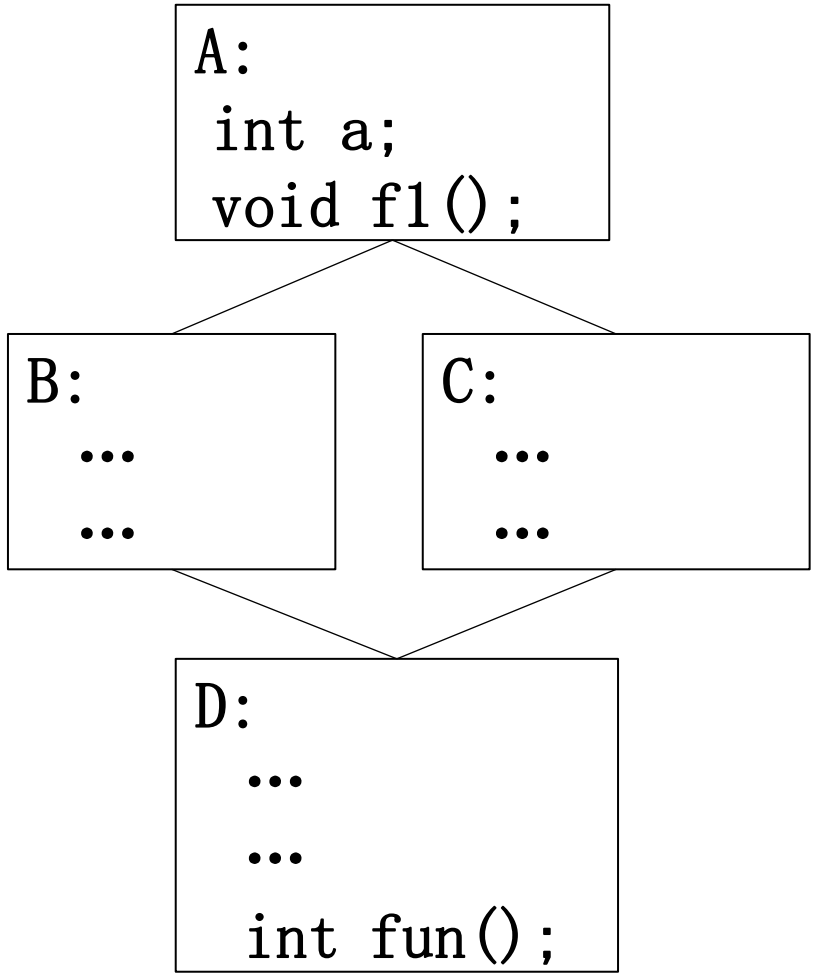
```
#include <iostream>
using namespace std;
static int x=0;
class A {
  public:
     int a;
     A() {
       a = ++x;
       cout << "A()" << endl; }
};
class B:public A {
  public:
     int b;
     B() {
       b = 20;
       cout << "B()" << endl; }
};
class C:public A {
  public:
     int c;
     C() {
       c = 30;
       cout << "C()" << endl; }
};
class D:public B, public C {
  public:
     int d;
     D(){
       d = 40;
       cout << "D()" <<endl; }
};
int main(){
  D d1;
  cout << sizeof(d1) << endl;
  int *p = (int *)&d1;
  cout << *p << endl;
  cout << *(p+1) << endl;
  cout << *(p+2) << endl;
  cout << *(p+3) << endl;
  cout << *(p+4) << endl;
  return 0;
}
```




```
A()
B()
A()
C()
D()
20
1
20
2
30
40
```

- 通过直接/间接方式继承同一个基类两个导致的同名：
  - 问题2：如何区分？

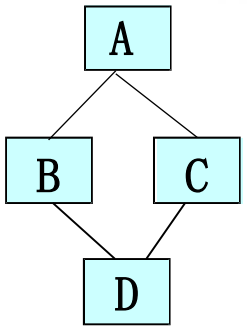
    通过可区分的类的作用域符来进行区分



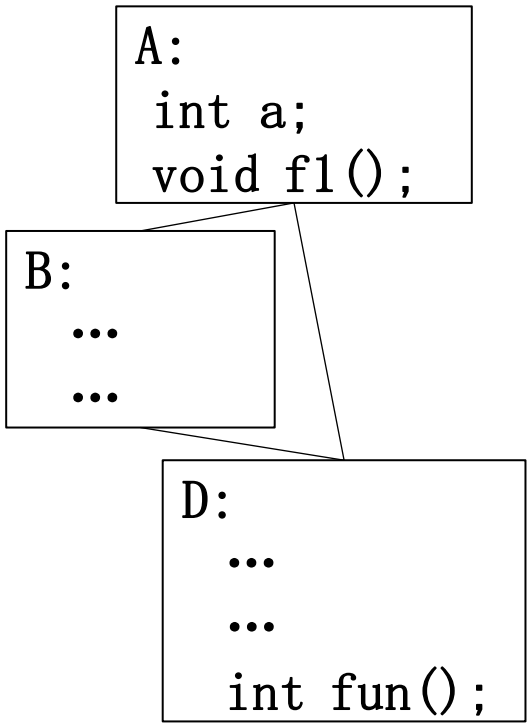

```
int D::fun()
{
    A::a = 5; //建议不用A::
    A::f1();  //建议不用A::
    B::a = 10;
    B::f1();
    C::a = 15;
    C::f1();
    return 0;
}
```

```
#include <iostream>
using namespace std;
static int x = 0;
class A {
public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl; }
};
class B :public A {
public:
    int b;
};
class C :public A {
public:
    int c;
};
class D :public B, public C {
public:
    int d;
    int fun();
};
```

```
int D::fun(){
    A::a = 5; //不建议
    A::f1();  //不建议
    B::a = 10;
    B::f1();
    C::a = 15;
    C::f1();
    return 0;
}
int main() {
    D d1;
    cout << sizeof(d1) << endl;
    d1.fun();
    d1.f1();   //报错：对f1的访问不明确
    d1.A::f1();//报错：对A的访问不明确
    d1.B::a=10;  但此句不同编译器会有区别
    d1.B::f1();
    d1.C::a=15;
    d1.C::f1();
    return 0;
}              //删除报错语句后的运行结果：
```




```
20
5
10
15
10
15
```



D对象的内存映象(两份A)

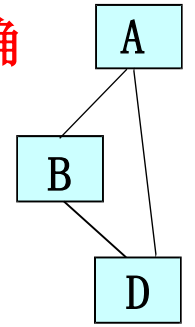
A的数据成员	B
B的数据成员	
A的数据成员	A
D的数据成员	D

```
#include <iostream>
using namespace std;
static int x=0;
class A {
  public:
    int a;
    A() {
      a = ++x;
      cout << "A()" << endl; }
};
class B :public A {
  public:
    int b;
    B() {
      b = 20;
      cout << "B()" << endl; }
};
```

```
class D :public B, public C {
  public:
    int d;
    D(){
      d = 40;
      cout << "D()" <<endl;}
};
int main(){
    D d1;
    cout << sizeof(d1) << endl;
    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```

```
A()
B()
A()
D()
16
1
20
2
40
```

```
#include <iostream>
using namespace std;
static int x = 0;
class A {
public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl; }
};
class B :public A {
public:
    int b;
};
class D :public B, public A {
public:
    int d;
    int fun();
};
int D::fun()
{
    A::a = 10; //不建议
    A::f1();   //不建议
    B::a = 15;
    B::f1();
    a = 15;    //报错：对a的访问不明确
    f1();      //报错：对f1的访问不明确
    return 0;
}
int main() {
    D d1;
    cout << sizeof(d1) << endl;
    d1.fun();
    d1.f1();   //报错：对f1的访问不明确
    d1.a = 15;  //报错：对a的访问不明确
    d1.A::a = 10; //不建议
    d1.A::f1();   //不建议
    d1.B::a = 15;
    d1.B::f1();
    return 0;
}
```



//删除报错语句后的运行结果：

16
10
15
10
15

目录

- 虚基类
  - 虚基类的形式
  - 虚基类的实例
  - 虚基类的构造函数

2.1 虚基类的形式

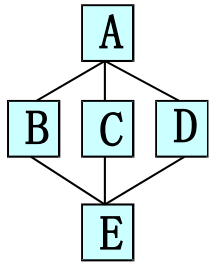
- 多重继承的问题：
  - 派生类中有多份相同的数据成员拷贝，占用较多的存储空间
  - 多个基类的相互交织可能会带来错综复杂的设计问题，命名冲突不可回避

➡引入虚基类，使相同基类只保留一份数据成员

- 虚基类声明的形式：

```
class 派生类名: virtual 存取限定符 基类名 {
    …
};
```

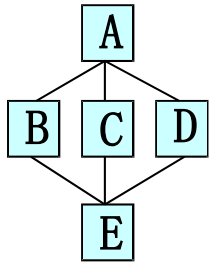
- 多重继承的构造函数调用




```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B:public A {
  public:
    B() {cout << "B()" << endl;}
};
class C:public A {
  public:
    C() {cout << "C()" << endl;}
};
class D:public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};
int main(){
    E e1;  //e1中有三份A的拷贝
}
```

```
A()
B()
A()
C()
A()
D()
E()
```

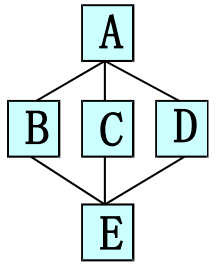
- 多重继承的构造函数调用：含虚基类




```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B: virtual public A {
  public:
    B() {cout << "B()" << endl;}
};
class C: virtual public A {
  public:
    C() {cout << "C()" << endl;}
};
class D: virtual public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};
int main(){
    E e1;  //e1中有一份A的拷贝
}
```

```
A()
B()
C()
D()
E()
```

- 多重继承的构造函数调用：含虚基类




```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B: virtual public A {
  public:
    B() {cout << "B()" << endl;}
};
class C: virtual public A {
  public:
    C() {cout << "C()" << endl;}
};
```

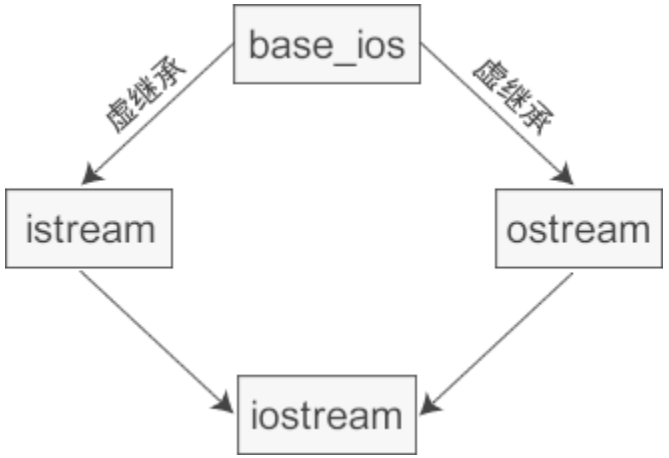
```
class D: public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};
int main(){
    E e1;  //e1中有两份A的拷贝
}
```

所有继承该基类的直接派生类都应声明为虚基类才能保证只有一份数据拷贝

```
A()
B()
C()
A()
D()
E()
```

2.2 虚基类的实例

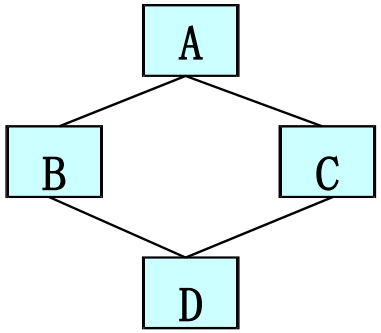
- C++标准库中的 iostream 类就是一个虚继承的实际应用案例



- iostream从istream和ostream直接继承而来，而istream和ostream又都继承自一个共同的名为base_ios的类，是典型的菱形继承
- istream和ostream必须采用虚继承，否则将导致iostream 类中保留两份 base_ios类的成员

2.3 虚基类的构造函数

- 构造函数的冲突问题：
  - 给基类自动传递信息时，存在多条不同的途径



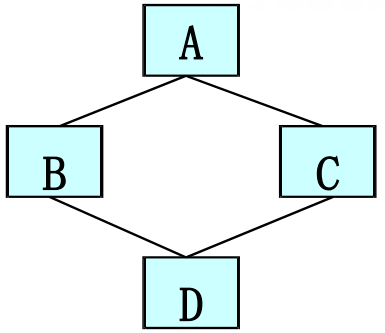
- 虚基类的构造函数：

  禁止信息通过中间类自动传递给基类

  - 使用虚基类时，必须在初始化列表最前方，显式的调用虚基类的构造函数；否则虚基类将调用默认构造函数初始化变量
  - 调用时，由派生类直接激活间接虚基类的构造函数，其直接基类不再自动激活虚基类的构造

- 禁止信息通过中间类自动传递给基类

```
class A {
   public:
      A() { cout << "A()" << endl; } // 无参构造
      A(int x) { cout << "A(" << x << ")" << endl; } //一参构造
};
class B:virtual public A {
   public:
     B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
   public:
     C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
   public:
      D(int x, int y, int z):A(x),B(y),C(z) {
               cout << "D()" << endl;}
};
```

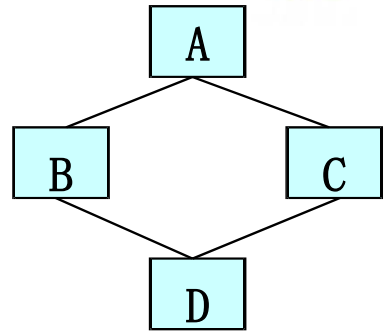



```
int main(){
  D d1(1,2,3);
  cout << endl;
  C c1(4);
  cout << endl;
  B b1(5);
}
```

```
A(1)
B()
C()
D()

A(4)
C()

A()
B()
```

- 禁止信息通过中间类自动传递给基类

```
class A {
   public:
      A() { cout << "A()" << endl; } // 无参构造
      A(int x) { cout << "A(" << x << ")" << endl; } //一参构造
};
class B:virtual public A {
   public:
     B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
   public:
     C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
   public:
      D(int x, int y, int z): B(y),C(z) {
               cout << "D()" << endl;}
};
```




```
int main(){
  D d1(1,2,3);
  cout << endl;
  C c1(4);
  cout << endl;
  B b1(5);
}
```

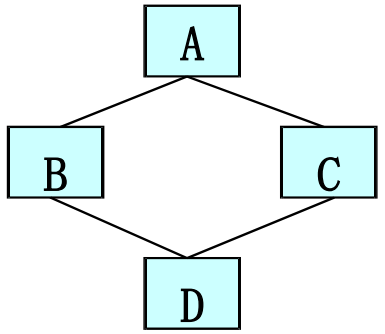
```
A()
B()
C()
D()

A(4)
C()

A()
B()
```

- 禁止信息通过中间类自动传递给基类

```
int main(){
  D d1(1,2,3);
  cout << endl;
  C c1(4);
  cout << endl;
  B b1(5);
}
```

➡ //d1对象生成时会自动激活A、B、C的构造函数，再调用D的构造函数。B/C的构造函数被调用时，不再自动激活A的构造函数

➡ //直接基类是虚基类时：与非虚基类一样，b1/c1生成时会自动激活A的无参/有参构造函数



目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

目录

- 赋值兼容规则
  - 赋值兼容规则的含义
  - 赋值兼容规则的使用未知

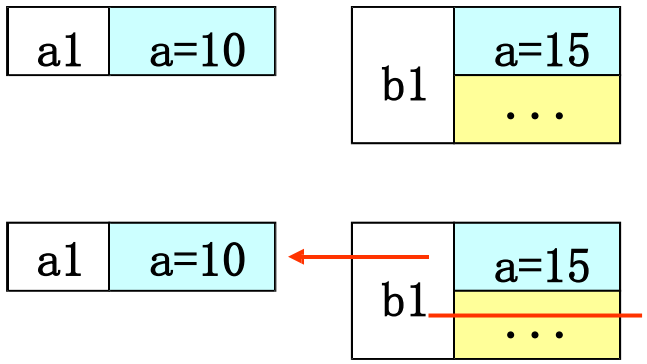
- 引例：基类与派生类的转换

```
#include <iostream>
using namespace std;
class A {
    public:
        int a;
};
class B:public A {
    public:
        int b;
};
```

```
int main()
{   A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl;
    b1 = a1;  //编译错!!!
    cout << a1.a << endl;
}
```

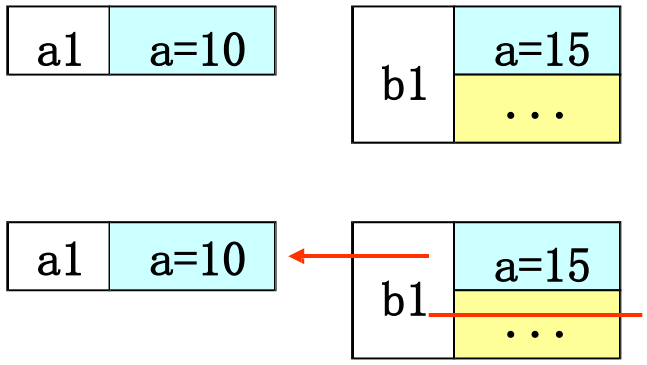
```
int main()
{   A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl; 10
    a1 = b1;
    cout << a1.a << endl; 15
}
```

赋值兼容规则:



3.1 赋值兼容规则的含义

- 赋值兼容规则：在需要基类对象的任何位置，均可以使用公有继承的派生类对象
  - ➢ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问
  - ➢ 将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃
    - ✓ 如果不希望直接对应拷贝，则可根据需要自行定义=重载或复制构造函数
    - ✓ 当基类中包含动态申请内存时，赋值兼容规则可能出错（参考=重载及复制构造函数）



- 派生类->基类：直接对应拷贝

```
class A {
    public:
        int a;
};
class B:public A {
    public:
      int b;
};
```

```
int main()
{
    B b1;
    b1.a = 15;
    A a1(b1); //复制构造
    A a2;
    a2 = b1;
    cout << a1.a << endl; //15
    cout << a2.a << endl; //15
}
```

- 派生类->基类：=重载

```
class B; //提前声明
class A {
  public:
      int a;
      A() {} //无参空体
      A(B &b);//不能体内实现
      A& operator=(B &b);//不能体内实现
};
class B:public A {
   public:
    int b;
};
A::A(B &b) //一参构造
{  a = b.a*2; }
```

```
A& A::operator=(B &b) //=重载
{   a = b.a*2;
    return (*this);
}
int main()
{
    B b1;
    b1.a = 15;
    A a1(b1), a2;
    //一参，无参构造
    a2 = b1; //= 重载
    cout << a1.a << endl; //30
    cout << a2.a << endl; //30
}
```

- 派生类->基类：动态内存申请 运行出错

```
class A {
   private:
     char *s;
   public:
     int a;
     A()
     {
        s = new char[20];
     }
     ~A()
     {
        cout << "A析构" << endl;
        delete s;
     }
};
```

```
class B:public A {
   public:
     int b;
     ~B() { cout << "B析构" << endl;}
};
int main()
{
   B b1;
   b1.a = 15;
   A a1(b1); //复制构造
}
```

3.2 赋值兼容规则的使用位置

1. 派生类对象可初始化基类的对象或引用

```
//示例：
class A {…};
class B:public A {…};

B b1;
A a1(b1);    //需要A对象，用B对象替代
A &a2 = b1; //需要A对象，用B对象替代
```

3.2 赋值兼容规则的使用位置

2. 派生类对象可出现在函数参数/返回值为基类的地方

```
//示例：
void f1(A  a1) { ... }  //形参为A对象
void f2(A &a1) { ... }  //形参为A对象的引用
void f3(A *pa) { ... }  //形参为A对象的指针
A f4(){ static B b1; return b1; //返回值为A对象 }
int main()
{    B b1;
     f1(b1);  //B对象出现在要求A对象的位置
     f2(b1);  //B对象出现在要求A对象引用的位置
     f3(&b1); //B对象出现在要求A对象指针的位置
}
```

3.2 赋值兼容规则的使用位置

3. 派生类对象可赋值给基类

```
//示例：
A a1;
B b1;
a1 = b1;   //需要A对象，用B对象替代
```

4. 派生类对象的指针可出现在基类指针出现的位置

```
//示例：
A *pa;
B b1, *pb = &b1;
pa = pb;  //需要A对象的地址，用B对象的地址替代
pa = &b1;  //需要A对象的地址，用B对象的地址替代
```

3.2 赋值兼容规则的使用位置

- 赋值兼容规则：
  - 赋值兼容是实现多态的一个前提（模块09）
  - 赋值兼容规则+指针或引用+虚函数==多态（模块09）

目录

- 继承和动态内存分配

4.1 继承和动态内存分配

- 引例

```
class baseDMA{    //基类使用了动态内存
private:
    char *label;
    int rating;
public:
    baseDMA(const char *lb = "null", int r = 0);
    baseDMA(const baseDMA &rs);                    //复制构造函数
    virtual ~baseDMA();                            //析构函数
    baseDMA & operator=(const baseDMA &rs);        //重载赋值运算符
…
};
```

4.1 继承和动态内存分配

- 情况1：派生类不使用new

```
class lacksDMA:public baseDMA{    //派生类
private:
    char color[40];
public:
…
};
```

不需要为lacksDMA类定义显示析构函数、复制构造函数和赋值运算符：

- lacksDMA成员不需执行任何特殊操作，所以默认析构函数是合适的
- lacksDMA类的默认复制构造函数使用显示baseDMA复制构造函数来复制lacksDMA对象的baseDMA部分，因此，默认复制构造函数也是合适的
- 默认的赋值构造函数将自动使用基类的赋值构造函数完成基类组件的赋值

4.1 继承和动态内存分配

- 情况2：派生类使用new

```
class hasDMA:public baseDMA{    //派生类
private:
    char *style;  //use new in constructors
public:
    ...
};
```

必须为hasDMA类定义显示析构函数、复制构造函数和赋值运算符！

4.1 继承和动态内存分配

- 情况2：派生类使用new
  - 显示析构函数

```
baseDMA::~baseDMA() //依赖于基类的析构函数来释放指针label管理的内存
{
   delete [] label;
}
hasDMA::~hasDMA()
{
   delete [] style;  //释放指针style管理的内存
}
```

4.1 继承和动态内存分配

- 情况2：派生类使用new
  - 复制构造函数

```
baseDMA::baseDMA(const baseDMA &rs){
    label = new char[strlen(rs.label) + 1];
    strcpy(label,rs.label);
    rating = rs.rating;
}
hasDMA::hasDMA(const hasDMA &hs):baseDMA(hs){
    style = new char[strlen(hs.style) + 1];
    strcpy(style,hs.style);
}
```

基类的指针或引用可以指向派生类的指针或引用

hasDMA复制构造函数只能访问hasDMA的数据，因此它必须调用baseDMA复制构造函数来处理baseDMA共享的数据

4.1 继承和动态内存分配

- 情况2：派生类使用new
  - 赋值运算符

```
baseDMA & baseDMA::operator=(const baseDMA &rs)  //基类
{
    if(this == &rs)
        return *this;
    delete [] label;
    label = new char[strlen(rs.label)+1];
    strcpy(label,rs.label);
    rating = rs.rating;
    return *this;
}
```

4.1 继承和动态内存分配

- 情况2：派生类使用new
  - 赋值运算符

```
hasDMA & hasDMA::operator=(const hasDMA &hs)    //派生类
{
        if(this == &hs)
            return *this;
        baseDMA::operator=(hs);   //copy base portion 函数表示法
        delete [] style;    //prepare for new style
        style = new char[strlen(hs.style)+1];
        strcpy(style,hs.style);
        return *this;
}
```

//含义为*this = hs; 运算符表示法
如按此写法，编译器将使用
hasDMA::operator=()，从而形成递归调用

4.1 继承和动态内存分配

- 情况2：派生类使用new（总结）
  - 显示析构函数：自动完成
  - 复制构造函数：在初始化成员列表中调用基类的复制构造函数完成；如果不这样做，将自动调用基类的默认构造函数
  - 赋值运算符：使用作用域解析运算符显示的调用基类的赋值运算符来完成
- 使用动态内存分配和友元的继承示例
  - primer书：集成baseDMA、lacksDMA、hasDMA类

    //dma.h

    //dma.cpp

    //usedma.cpp

总结

- 多重继承（熟悉）
  - 多重继承的声明及使用
  - 多重继承的派生类
  - 多重继承引起的二义性问题
- 虚基类（熟悉）
  - 虚基类的形式
  - 虚基类的实例
  - 虚基类的构造函数
- 赋值兼容规则（熟悉）
  - 赋值兼容规则的含义
  - 赋值兼容规则的使用未知
- 继承和动态内存分配（了解）