School of Artificial Intelligence
Jilin University

C++面向对象程序设计

人工智能 学院

马锐

个人介绍

SFU SIMON FRASER UNIVERSITY Computing Science

- **Ph.D.（2013年至2018年)**
  - 研究方向：计算机图形学，3D几何建模和形状分析等

吉林大学 数学学院

- **硕士（2010年至2013年)**
  - 吉林大学，计算数学
- **本科（2006年至2010年)**
  - 吉林大学，信息与计算科学

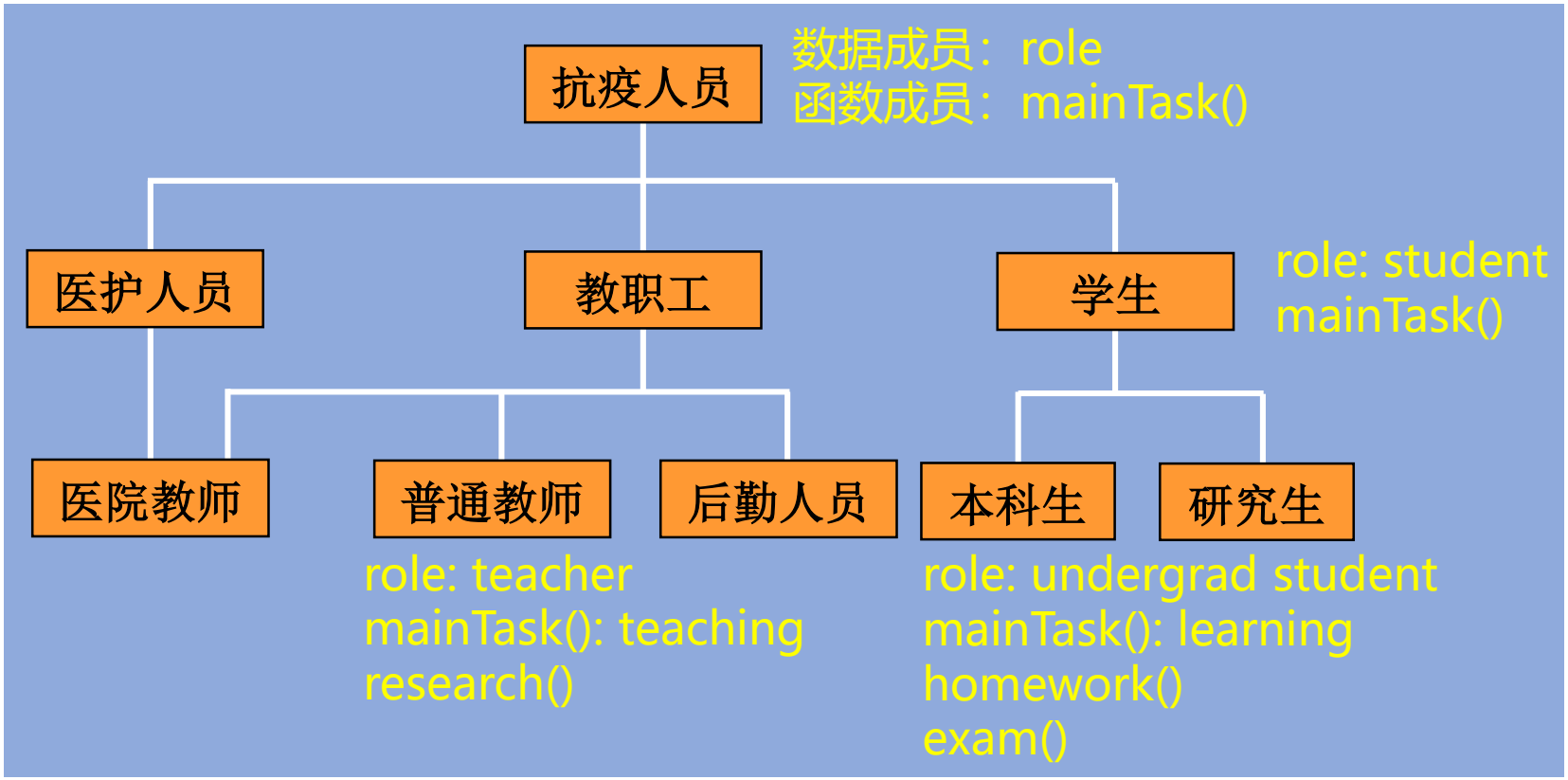
- **2021.10 – 至今，准聘副教授，人工智能学院**
  - 研究方向：基于人工智能的2D和3D智能可视内容的**分析** 、**生成**和**应用**
- **2019.08 – 2021.08，Senior Researcher，华为加拿大研究院温哥华研究所海思实验室(HiSilicon Lab)**
  - 人体步态识别， 3D物体建模研究
- **2018.01 – 2019.08，Research Scientist，Altumview Systems, Canada**
  - 人脸识别、跟踪
  - 跌倒检测
  - 室内机器人对话、导航

关于提问的几点说明

- 听讲的过程中，建议自己用Word文档记录没有听懂的地方（和页码），在课堂余留的提问环节里，将问题输入到聊天框里，由我统一进行回答

- 在讲课的过程中，我会对一些知识点进行提问，由大家志愿（voluntary）进行回答，想参与回答的同学，在聊天框内进行报名，由我随机选一个进行回答

继承与派生：以抗疫分工为例



课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

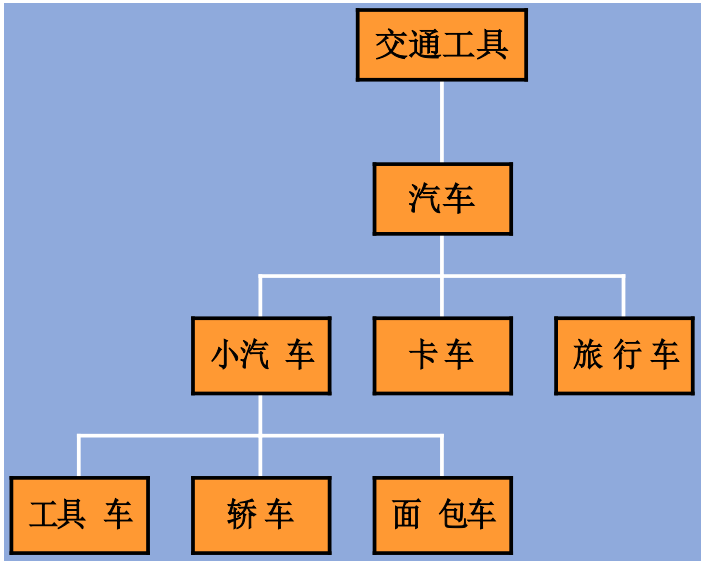
类型兼容性规则

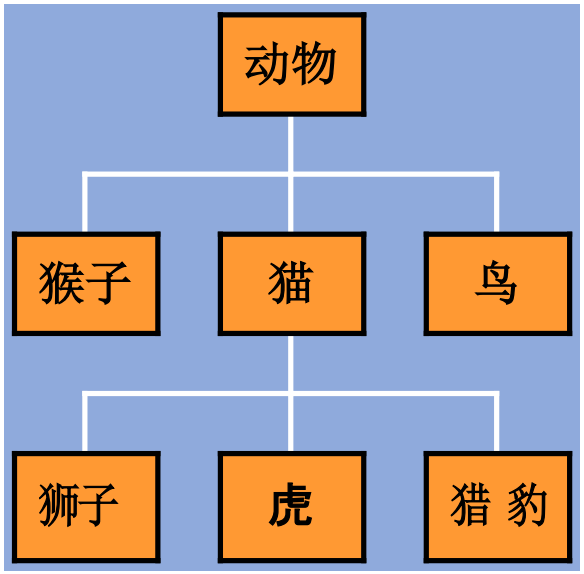
构造函数、析构函数

类成员的标识与访问

7.1 类的继承与派生

- 保持已有类的特性而构造新类的过程称为**继承 (Inheritance)**。
- 从另一个角度来看，在已有类的基础上新增自己的特性而产生新类的过程称为**派生 (Derive)**。
- 被继承的已有类称为**基类 (Base class)**（或父类）。
- 派生出的新类称为**派生类 (Derived class)**（或子类）。





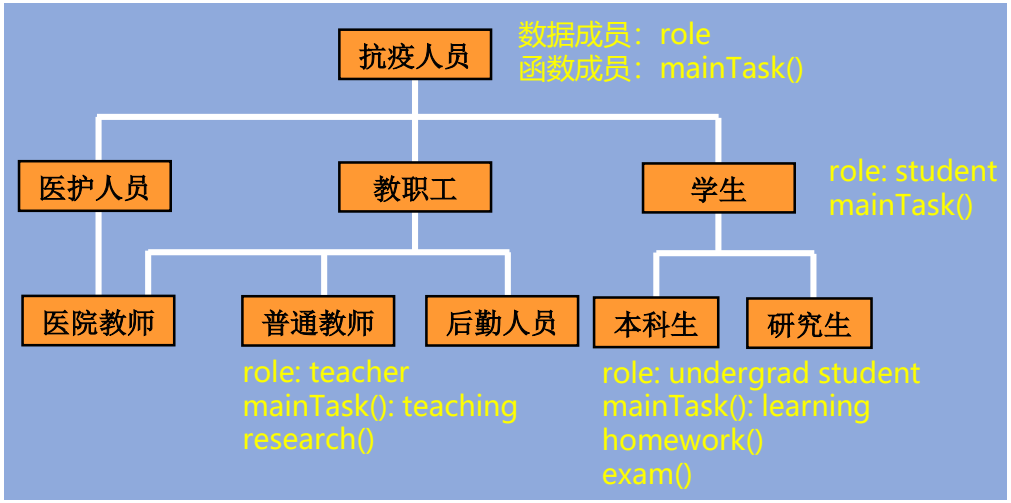
- 继承的目的：实现代码重用，希望尽量利用原有的类。
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，在已有类的基础上新增自己的特性。
- 类的派生实际是一种演化、发展的过程，即通过扩展、更改和特殊化，从一个已知类出发建立一个新类。

派生类的定义

class 派生类名：继承方式1 基类名1 继承方式2 基类名2 … {
    派生类成员声明;
}

继承方式包括：公有继承、保护继承和私有继承

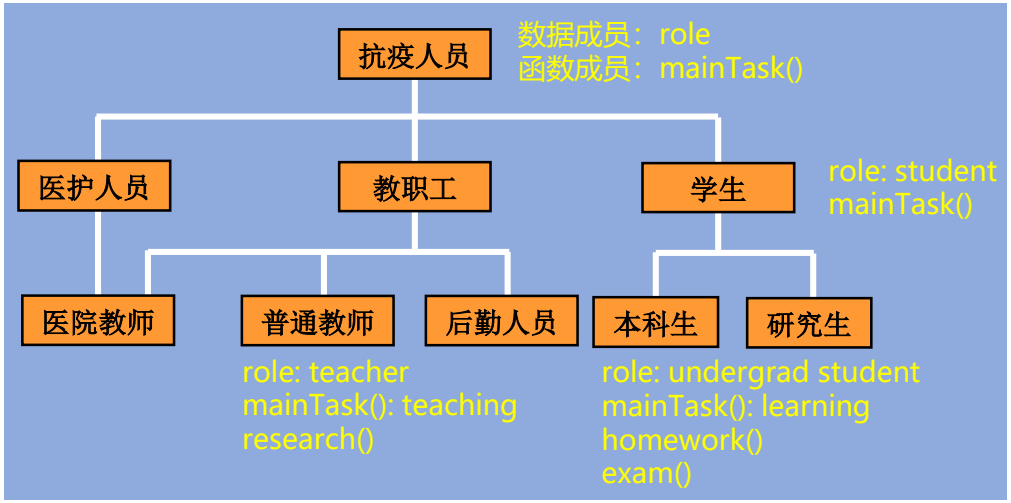
派生类成员包括：从基类继承来的所有成员（构造函数和析构函数除外）和新增的数据成员、函数成员




```
class UnderGrad: public Student
{
    void homework(); // 新增数据成员
    void exam(); // 新增函数成员
}
```

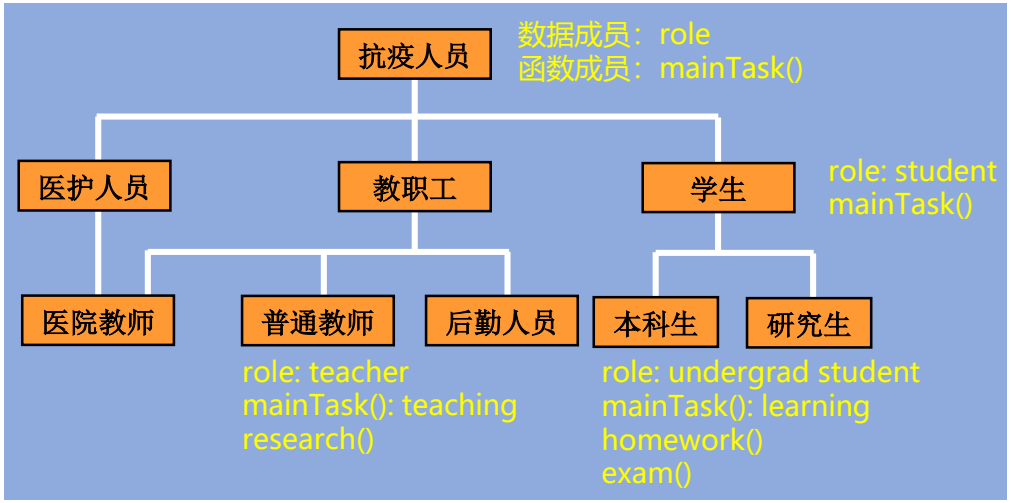
派生类的定义

- 通过派生，可以形成一个**类族**
  - 派生出来的新类可以作为基类再继续派生新的类
  - 一个基类也可以同时派生出多个派生类
- 直接参与派生出某类的基类称为**直接基类**
- 基类的基类或更高层基类称为**间接基类**



派生新类的过程

1. 吸收基类成员
   - 数据成员和函数成员
   - 构造函数和析构函数除外
2. 改造基类成员
   - 通过继承方式 (public, protected, private)，控制**基类成员的访问控制权限**
   - 在派生类中声明和基类数据或者函数同名的成员 → **同名隐藏**
3. 添加新的成员
   - 根据实际情况需要，添加数据或者函数成员，实现必要的新增功能



同名隐藏举例：基类和派生类中同时定义了mainTask()函数，但具体实现可以不同

课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

继承方式

- 不同继承方式的影响主要体现在：
  - 派生类成员对基类成员的访问权限
  - 通过派生类对象对基类成员的访问权限

- 三种继承方式
  - 公有继承 (public)
  - 私有继承 (private)
  - 保护继承 (protected)

公有继承 (public)

- 基类的 public 和 protected成员都出现在派生类中，其访问属性在派生类中保持不变，在派生类中可以直接访问它们，但基类的private成员不可直接访问。

- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。

- 通过派生类的对象只能访问基类的public成员。

公有继承 (public) 举例

```
class Point    //基类Point类的声明
{
public:   //公有函数成员
    void InitP(float xx=0, float yy=0)
    {X=xx;Y=yy;}

    void Move(float xOff, float yOff)
    {X+=xOff;Y+=yOff;}

    float GetX() {return X;}

    float GetY() {return Y;}

private: //私有数据成员
    float X,Y;
};
```

```
class Rectangle: public Point  //派生类声明
{
public:    //新增公有函数成员
    void InitR(float x, float y, float w, float h)
    {
        InitP(x,y); //调用基类公有成员函数
        W=w;H=h;
        }
    float GetH() {return H;} //新增私有函数成员
    float GetW() {return W;}
    void print();
private:   //新增私有数据成员
    float W,H;
};

void Rectangle::print(){
        //cout<<X<<" "<<Y<<endl;
        //错误   不能直接访问基类私有成员
        cout<< GetX() <<" "<<GetY()<<endl;
        //正确 可以通过基类公有成员函数间接访问
}
```

为什么错误/正确?

公有继承 (public) 举例

```
#include<iostream>
#include<cmath>
using namecpace std;
int main()
{  Rectangle rect;
      rect.InitR(2,3,20,10);
      rect.Move(3,2);   //派生类对象访问基类公有函数成员
      cout<<rect.GetX()<<   ','    //派生类对象通过基类公有函数成员
访问私有数据成员
          <<rect.GetY()<<','
            <<rect.GetH()<<','
            <<rect.GetW()<<endl;
      return 0;
}
```

私有继承 (private)

- 基类的public和protected成员都以private身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员。

私有继承 (private) 举例

```
class Point//基类Point类的声明
{public:  //公有函数成员
      void InitP(float xx=0, float yy=0)
      {X=xx;Y=yy;}
      void Move(float xOff, float yOff)
      {X+=xOff;Y+=yOff;}
      float GetX() {return X;}
      float GetY() {return Y;}

private:  //私有数据成员
      float X,Y;
};
```

```
class Rectangle: private Point//派生类声明
{public:  //新增外部接口
      void InitR(float x, float y,
       float w, float h){
      InitP(x,y); W=w;H=h;
      }//访问基类公有成员
      float GetX() {return Point::GetX();}
      float GetY() {return Point::GetY();}
      float GetH() {return H;}
      float GetW() {return W;}
private:  //新增私有数据
      float W,H;
};
```

私有继承 (private) 举例

```
#include<iostream>
#include<cmath>
using namecpace std;
int main()

{  Rectangle rect;
      rect.InitR(2,3,20,10);
      //rect.Move(3,2);   //错误，由于private继承，使用派生类对象rect已经不能访问基类中的原公有成员
      cout<<rect.GetX()<<','<<rect.GetY()<<','
            <<rect.GetH()<<','<<rect.GetW()<<endl;
      return 0;
}
```

- 在私有继承的情况下，派生类的对象不能访问到任何一个基类的成员，为了使用基类中的一部分外部接口，则需要在派生类中**重新声明同名的成员**
- 一般情况下私有继承的使用比较少

保护继承 (protected)

- 基类的public和protected成员都以protected身份出现在派生类中，但基类的private成员不可直接访问。

- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。

- 通过派生类的对象不能直接访问基类中的任何成员

protected 成员特点与作用

- 对建立其所在类对象的模块来说，它与 private 成员的性质相同。

- 对于其派生类（即对于派生类的成员函数）来说，它与 public 成员的性质相同。

- 既实现了数据隐藏，又方便继承，实现代码重用。

例7-3 protected 成员举例

```
class A {
    protected:
            int x;
}
int main()
{
    A a;
    a.x=5;    //错误
}
```

为什么错误?

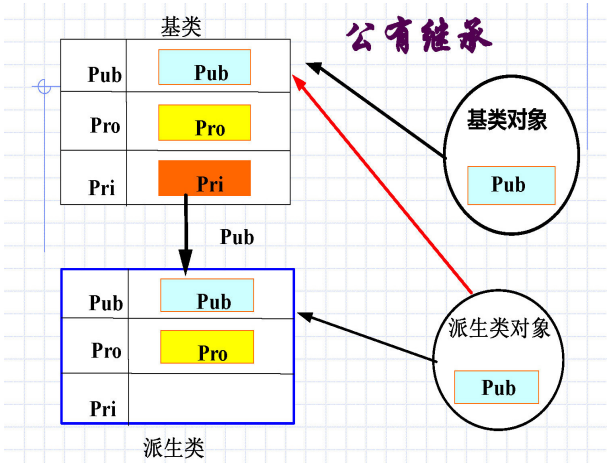
类的对象不能直接访问protected成员

```
class A {
    protected:
            int x;
}
class B: protected A{
    public:
        void Function();
};
void B::Function()
{
    x=5;   //正确
}
```

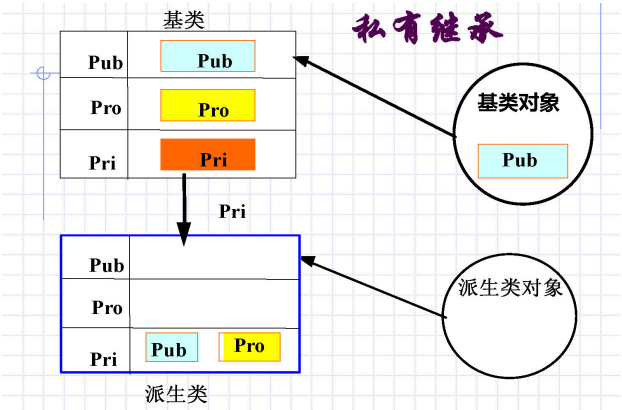
为什么正确?

用protected继承得到的派生类中，可以使用成员函数访问基类的protected成员

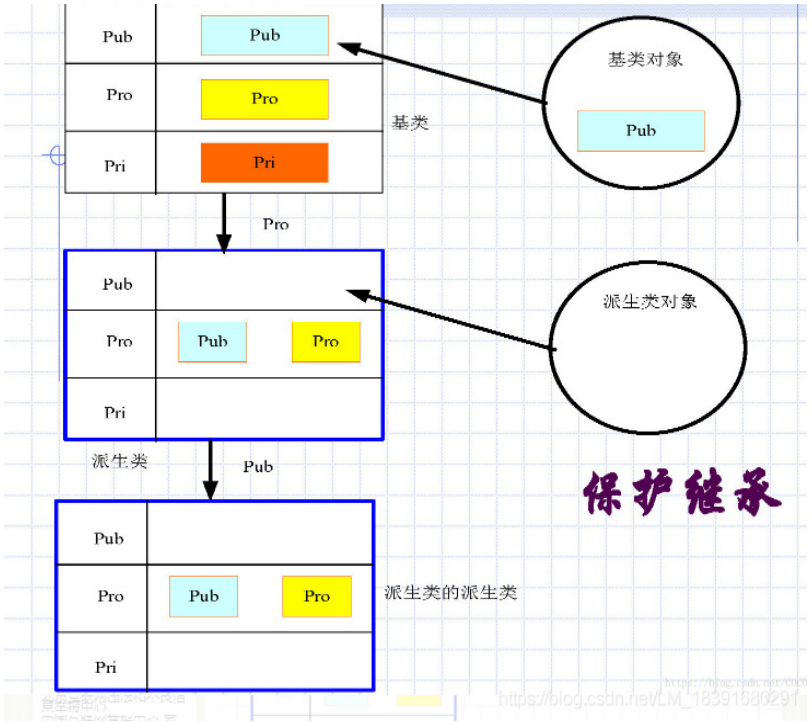
继承方式总结



派生类内部可直接访问Pub和Pro（均保持原有属性），派生类对象只能直接访问Pub



派生类内部可直接访问Pub和Pro（均变为private），派生类对象不能直接访问任何成员



派生类内部可直接访问Pub和Pro（均变为protected），派生类对象不能直接访问任何成员

所有继承方式中，派生类的内部和派生类的对象均不能直接访问基类的Pri成员

继承与组合的区别

```
class Wheel {                    //轮子类
public:
    void roll();                 //轮子转动
    …
};
class Automobile {               //汽车类
public:
    void move();                 //汽车移动
    …
private:
    Engine engine;               //汽车引擎
    Wheel wheels[4];             //4 个车轮
    …
};
```

类的组合反应的是 类之间有“has-a”（有一个）的关系；
也可以理解为整体与部分的关系

```
class Truck: public Automobile {          //卡车
public:
    void load(…);                         //装货
    void dump(…);                         //卸货
private:
    …
};
class Pumper: public Automobile {         //消防车
public:
    void water();                         //喷水
private:
    …
};
```

类的公有继承反应的是 类之间有“is-a”（是一个）的关系；
也可以理解为特殊与一般的关系

注意：私有继承和保护继承不一定表示“is-a”的关系，具体参看课本7.8.1小节

课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

Q&A

课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

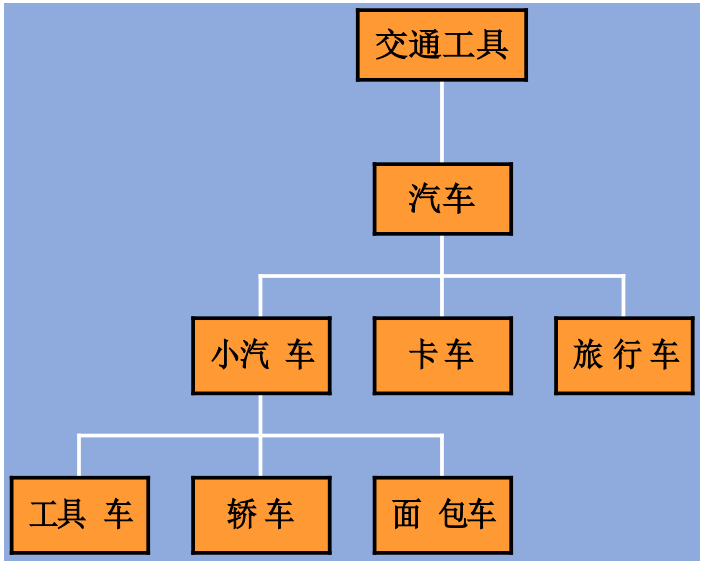
类型兼容性规则

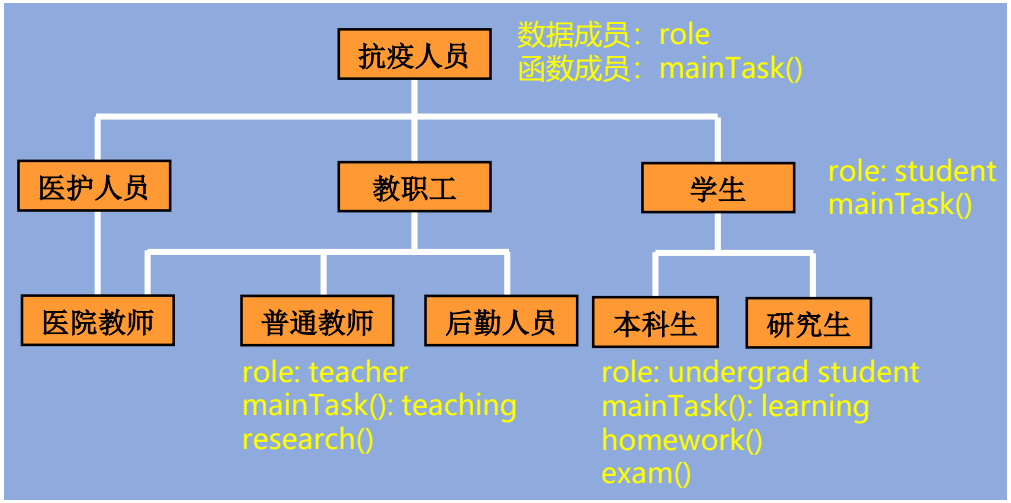
构造函数、析构函数

类成员的标识与访问

单继承与多继承

- 单继承
  - 派生类只从一个基类派生
- 多继承
  - 派生类从多个基类派生
- 多重派生
  - 由一个基类派生出多个不同的派生类
- 多层派生
  - 派生类又作为基类，继续派生新的类





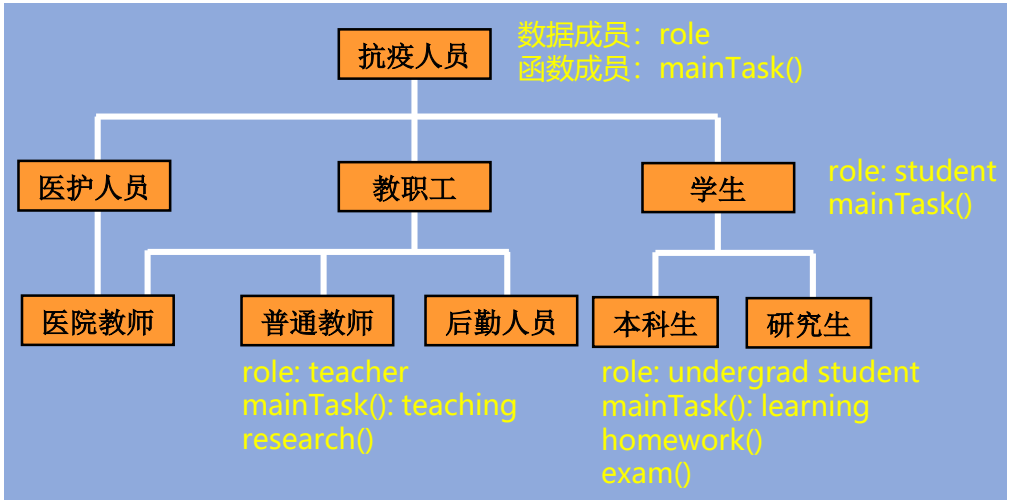
多继承派生类的声明

class 派生类名：继承方式1 基类名1，
继承方式2 基类名2，…
{
成员声明;
}

注意：

- 每一个“继承方式”，只用于限制对紧随其后之基类的继承
- 当没有显式指定继承方式时，默认为private

class MedicalTeacher: public MedicalPersonel, JLUPersonel
{…}



多继承举例

```
class A{
    public:
        void setA(int);
        void showA();
    private:
        int a;
};
class B{
    public:
        void setB(int);
        void showB();
    private:
        int b;
};
class C : public A, private B
{
    public:
        void setC(int, int, int);
        void showC();
    private:
        int c;
};
```

多继承举例

```
void  A::setA(int x)
{   a=x;  }
void B::setB(int x)
{   b=x;  }
void C::setC(int x, int y, int z)
{   //派生类成员直接访问基类的公有成员
    setA(x);
    setB(y);
    c=z;
}
//其他函数实现略
```

```
int main() {
    C obj;
    obj.setA(5);
    obj.showA();
    obj.setC(6,7,9);
    obj.showC();
    // obj.setB(6);  //错误，C是B的私有继承，setB(int)变为C的私有成员
    // obj.showB(); //错误
    return 0;
}
```

课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

类型兼容规则

- 一个公有派生类的对象在使用时可以被当作基类的对象，反之则禁止。换句话说，在需要使用基类对象的任何地方，都可以使用公有派生类的对象来替代。

```
Class B {...}
Class D: public B{...}
B b1, *pb1;
D d1;
```

- 具体表现在：
  - 派生类的对象可以被赋值给基类对象（隐含转换）
  - 派生类的对象可以初始化基类的引用
  - 指向基类的指针也可以指向派生类

```
b1 = d1;
B &rb = d1;
pb1 = &d1;
```

- 注意：通过基类对象名、指针只能使用从基类继承的成员。

例7-4 类型兼容规则举例

```
#include <iostream>
using namecpace std;
class B0 //基类B0声明
{ public:
      void display(){cout<<"B0::display()"<<endl;}   //公有成员函数
};
class B1: public B0
{
 public:
      void display(){cout<<"B1::display()"<<endl;}
};
class D1: public B1
{
public:
      void display(){cout<<"D1::display()"<<endl;}
};
void fun(B0 *ptr)
{     ptr->display();    //"对象指针->成员名"  }
```

7.4 类型兼容性规则

```
void main()  //主函数
{     B0 b0;   //声明B0类对象
      B1 b1;   //声明B1类对象
      D1 d1;   //声明D1类对象
      B0 *p;   //声明B0类指针

      p=&b0; //B0类指针指向B0类对象
      fun(p);

      p=&b1; //B0类指针指向B1类对象
      fun(p);

      p=&d1; //B0类指针指向D1类对象
      fun(p);
}
```

运行结果：？

B0::display()
B0::display()
B0::display()

通过这个例子可以看出，根据类型兼容规则，可以在基类对象出现的场合使用公有派生类的对象进行替代，但是替代之后派生类仅仅发挥出基类的作用。在学习完第8章的多态性之后，可以实现在保证类型兼容性的前提下，基类、派生类可以以不同的方式来实现同一个名字命名的函数。

拓展-基类向派生类转换及其安全性问题

基类指针向派生类指针的转换

- 派生类指针允许隐式转换为基类指针，因为它是安全的转换。基类指针转换成派生类指针，一定要显示转换。

```
Base * pb= new Derived();                    //将 Derived 指针隐含转换为 Base 指针
Derived * pd= static_cast< Derived * > (pb);  //将 Base 指针显式转换为 Derived 指针
```

合法，但不一定安全

基类对象向派生类对象转换

- 基类对象一般无法被显示转换成派生类对象，除非派生类有接收基类对象的构造函数，否则为非法操作。

```
Base b;
Derived d= static_case< Derived> (b);
```

只有在Derived类中定义了接收Base类的构造函数时合法

如何判断指针转换的安全性？（更多介绍请自行阅读课本7.8.3小节）
从特殊指针（如派生类指针）换到一般指针（如基类指针，void指针）是安全的；从一般指针到特殊指针的转换是不安全的。

课程纲要

① **第七章**

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

派生类的构造函数

- 基类的构造函数不被继承，派生类中需要根据情况声明自己的构造函数:
  - 如果对于基类初始化时，需要调用基类含有形参的构造函数，则**必须声明**派生类的构造函数，将参数传递给基类的构造函数。
  - 如果不需要调用基类的带参数的构造函数，也不需要调用新增的成员对象的带参数的构造函数，**可以不显式声明**派生类的构造函数，系统会隐含生成一个默认构造函数，该函数会使用基类的默认构造函数对继承自基类的数据进行初始化，然后调用派生类的默认构造函数。

派生类的构造函数

- 声明构造函数时，需要对基类的成员和本类中新增成员进行初始化。
- 由于派生类对于基类的很多成员对象是不能直接访问的（参见类成员的访问控制），对继承来的基类成员的初始化，需要调用基类构造函数完成。
- 派生类的构造函数需要给基类的构造函数传递参数。

单一继承时的构造函数

派生类名::派生类名(基类所需的形参，本类成员所需的形参):基类名(基类的参数表)，成员对象名1（成员对象1的参数表），成员对象名2（成员对象2的参数表）...｛

本类成员初始化赋值语句；

};

构造函数的调用次序：

- 调用基类构造函数
- 调用派生类数据成员对象的构造函数，调用顺序按照它们在类中声明的顺序
- 派生类的构造函数体中的内容

单一继承时的构造函数举例

```
#include<iostream>
using namespace std;
class B{
    public:
        B();
        B(int i);
        ~B();
        void Print() const;
    private:
        int b;
};
```

单一继承时的构造函数举例

```
B::B()
{  b=0;
   cout<<"B's default constructor called."<<endl;
}
B::B(int i)
{  b=i;
    cout<<"B's constructor called." <<endl;
}
B::~B()
{  cout<<"B's destructor called."<<endl;  }
void B::Print() const
{  cout<<b<<endl;  }
```

单一继承时的构造函数举例

```
class C:public B
{
    public:
      C();
     C(int i,int j);
     ~C();
     void Print() const;
   private:
     int c;
};
C::C()
{ c=0;
  cout<<"C's default constructor called."<<endl;
}
```

单一继承时的构造函数举例

```
C::C(int i,int j):B(i)
{ c=j;
  cout<<"C's constructor called."<<endl;
}
C::~C()
{ cout<<"C's destructor called."<<endl;
}
void C::Print() const
{ B::Print();   cout<<c<<endl;
}
void main()
{ C obj(5,6); obj.Print();
}
```

运行结果

```
B's constructor called.
C's constructor called.
5
6
C's destructor called.
B's destructor called
```

多继承时的构造函数

派生类名::派生类名(基类1形参，基类2形参，...基类n形参，本类形参):基类名1(基类1的参数),基类名2(基类2的参数),...基类名n(基类n的参数),成员对象名1（成员对象1的参数表），成员对象名2（成员对象2的参数表）...

```
{
        本类成员初始化赋值语句;
};
```

多继承时的构造函数

- 当基类中声明有默认形式的构造函数或未声明构造函数时，派生类构造函数可以不向基类构造函数传递参数。
- 若基类中未声明构造函数，派生类中也可以不声明，全采用默认缺省 (default) 形式的构造函数。
- 当基类声明有带形参的构造函数时，派生类也应声明带形参的构造函数，并将参数传递给基类构造函数。

多构造函数的调用次序

- 调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。
- 调用成员对象的构造函数，调用顺序按照它们在类中声明的顺序。
- 派生类的构造函数体中的内容。

复制构造函数

- 若用户没有编写显式的复制构造函数，系统在建立派生类对象时会自动生成一个默认复制构造函数，这个函数将：1）自动调用基类的默认复制构造函数；2）对派生类新增的成员对象一一执行复制

- 若用户想要编写派生类的复制构造函数，则需要为基类相应的复制构造函数传递参数。例如:

```
C::C(C &c1):B(c1)  {...}
```

注意：这里将派生类对象的引用c1传递给了基类作为参数，为什么?

这是因为类型兼容性规则允许用公有派生类的对象去初始化基类的引用

例7-5 派生类构造函数举例

```
#include <iostream>
using namespace std;
class B1        //基类B1，构造函数有参数
{public:
  B1(int i) {cout<<"constructing B1 "<<i<<endl;}
};
class B2        //基类B2，构造函数有参数
{public:
  B2(int j) {cout<<"constructing B2 "<<j<<endl;}
};
class B3        //基类B3，构造函数无参数
{public:
   B3(){cout<<"constructing B3 *"<<endl;}
};
```

例7-5 派生类构造函数举例

```
class C: public B2, public B1, public B3 // 注意继承的顺序
{
public://派生类的公有成员
   C(int a, int b, int c, int d):
       B1(a),memberB2(d),memberB1(c),B2(b)  {}
private:          //派生类的私有对象成员，注意成员声明的顺序
   B1 memberB1;
   B2 memberB2;
   B3 memberB3;
};
void main()
{  C obj(1,2,3,4);  }
```

运行结果：

```
constructing B2 2 // 对应B2(b)
constructing B1 1 // 对应B1(a)
constructing B3 * //基类B3的默认构造函数B3()
constructing B1 3 // 对应memberB1(c)
constructing B2 4 //对应memberB2(d)
constructing B3 * // member B3的默认构造函数B3()
```

派生类的析构函数

- 派生类过程中，析构函数也不被继承，如果需要对派生类进行显式析构的话，需要派生类自行声明析构函数
- 声明方法与一般（无继承关系时）类的析构函数相同。
- 不需要显式地调用基类的析构函数，系统会自动隐式调用。
- 析构函数的调用次序与构造函数相反。

注意：对于系统自动生成的默认的析构函数，虽然函数体是空的，但是并非不做任何事情，它会隐含地：1）调用派生类对象成员的析构函数；2）调用基类的析构函数

例7-6 派生类析构函数举例

```
#include <iostream>
using namecpace std;
class B1          //基类B1声明
{ public:
   B1(int i)   {cout<<"constructing B1 "<<i<<endl;}
   ~B1() {cout<<"destructing B1 "<<endl;}
};
class B2          //基类B2声明
{public:
   B2(int j)    {cout<<"constructing B2 "<<j<<endl;}
   ~B2() {cout<<"destructing B2 "<<endl;}
};
class B3          //基类B3声明
{public:
   B3(){cout<<"constructing B3 *"<<endl;}
   ~B3() {cout<<"destructing B3 "<<endl;}
};
```

```
class C: public B2, public B1, public B3
{
  public:
        C(int a, int b, int c, int d):
        B1(a),memberB2(d),memberB1(c),B2(b){}

  private:
        B1 memberB1;
        B2 memberB2;
        B3 memberB3;
};
void main()
{
  C obj(1,2,3,4);
}
```

析构函数的执行顺序?

运行结果

```
constructing B2 2
constructing B1 1
constructing B3 *
constructing B1 3
constructing B2 4
constructing B3 *

destructing B3
destructing B2
destructing B1
destructing B3
destructing B1
destructing B2
```

课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

Q&A

课程纲要

① **第七章**

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

同名隐藏规则

当派生类与基类中有相同成员时：

- 若未显式指名，则通过派生类对象使用的是派生类中的同名成员，即基类同名成员被隐藏。
- 如要通过派生类对象访问基类中被覆盖的同名成员，应使用基类名和作用域分辨符::进行限定。

注意：若派生类中声明了与基类成员函数同名的新函数，即使函数的形参不同，从基类继承的同名函数的所有重载形式也都会被隐藏

例7-7 多继承同名隐藏举例

```
#include <iostream>
using namecpace std;
class B1   //声明基类B1
{ public:  //外部接口
    int nV;
    void fun()  {cout<<"Member of B1"<<endl;}
};
class B2   //声明基类B2
{ public:  //外部接口
    int nV;
    void fun(){cout<<"Member of B2"<<endl;}
};
class D1: public B1, public B2
{ public:
    int nV; //同名数据成员
    void fun(){cout<<"Member of D1"<<endl;}   //同名函数成员
};
```

```
void main()
{ D1 d1;
  d1.nV=1;  //对象名.成员名标识, 访问D1类成员
  d1.fun();

  d1.B1::nV=2;//作用域分辨符标识, 访问基类B1成员
  d1.B1::fun();

  d1.B2::nV=3;//作用域分辨符标识, 访问基类B2成员
  d1.B2::fun();
}
```

运行结果

```
Member of D1
Member of B1
Member of B2
```

二义性问题

- 在多继承时，基类与派生类之间，或基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用虚函数（第8章）或同名隐藏规则来解决。
- 当派生类从多个基类派生，而这些基类又从同一个基类派生（多层继承），则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。

二义性问题举例（多继承）

```
class A
{
    public:
        void  f();
};
class B
{
    public:
        void f();
        void g();
};
```

```
class C: public A, public B { public:
        void g();
        void h();
};
```

如果声明：C c1;

则 c1.f(); 具有二义性

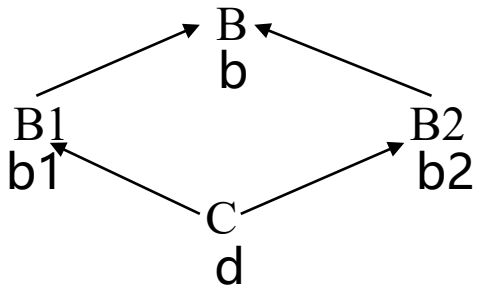
而 c1.g(); 无二义性（同名覆盖）

解决方法?

- **解决方法一**：用类名和作用域标识符来限定
  c1.A::f() 或 c1.B::f()
- **解决方法二**：同名覆盖
  在C 中声明一个同名成员函数f()，f()再根据需要调用 A::f() 或 B::f()

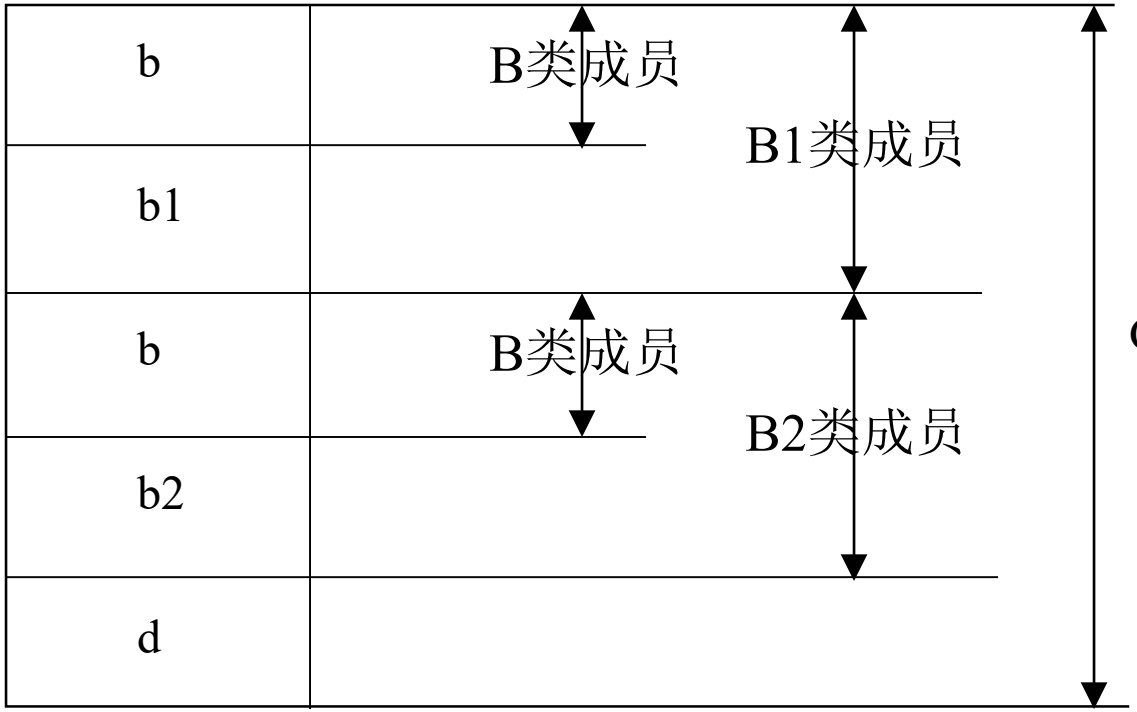
二义性问题举例（多层继承）

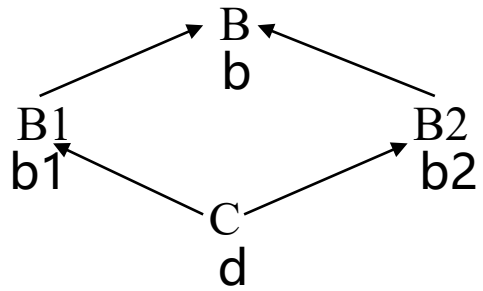
```
class B
{        public:
           int b;
}
class B1 : public B
{
      private:
           int b1;
}
class B2 : public B
{
      private:
           int b2;
};
```




```
class C : public B1,public
  B2
{
      public:
          int f();
      private:
          int d;
}
```

派生类C的对象的存储结构示意图



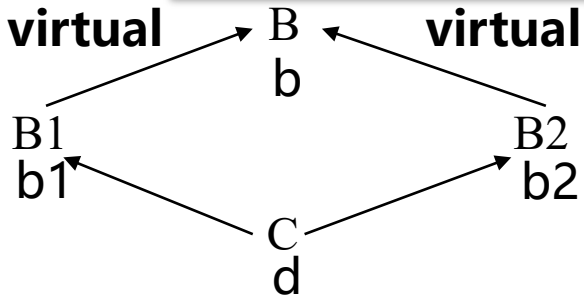


C类对象

有二义性:
C c;
c.b
c.B::b

无二义性:
c.B1::b
c.B2::b

虚基类



虚基类

- 主要用来解决多层继承时可能发生的对同一基类继承多次而产生的二义性问题
- 用于有共同基类的场合，从不同路径继承过来的同名数据成员和函数成员将在内存中具有唯一性
- 为最远的派生类提供唯一的基类成员，而不重复产生多次拷贝，虚基类的成员和派生类一起维护一个同一个内存数据副本

声明

- 以virtual修饰说明基类例：class B1:virtual public B, public A
- virtual关键字只对紧跟其后的基类起作用

注意

- 一般情况下，在第一级继承时就要将共同基类设计为虚基类

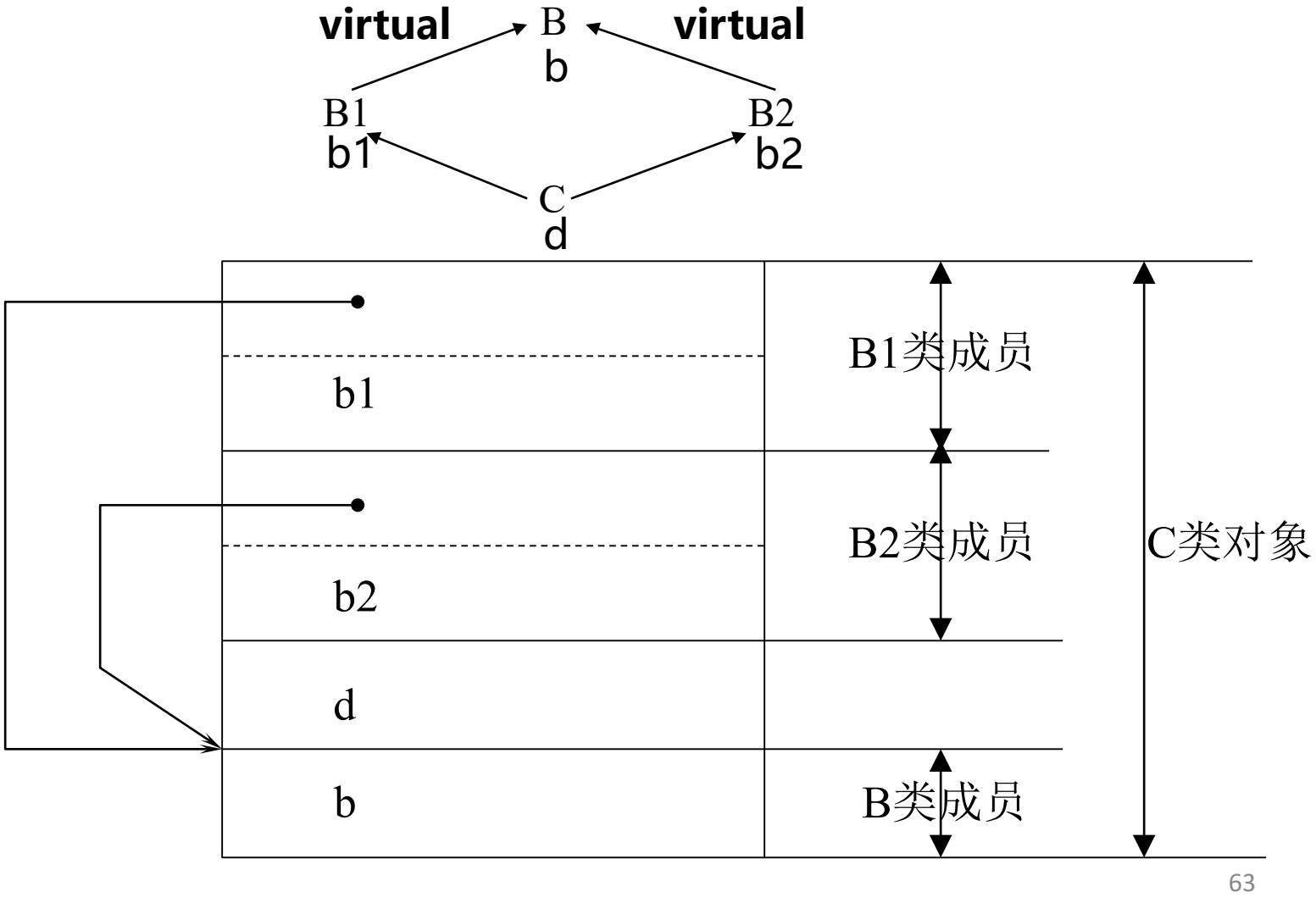
虚基类举例

```
class B{ private: int b;};
class B1 : virtual public B { private: int b1;};
class B2 : virtual public B { private: int b2;};
class C : public B1, public B2{ private: float d;}
```

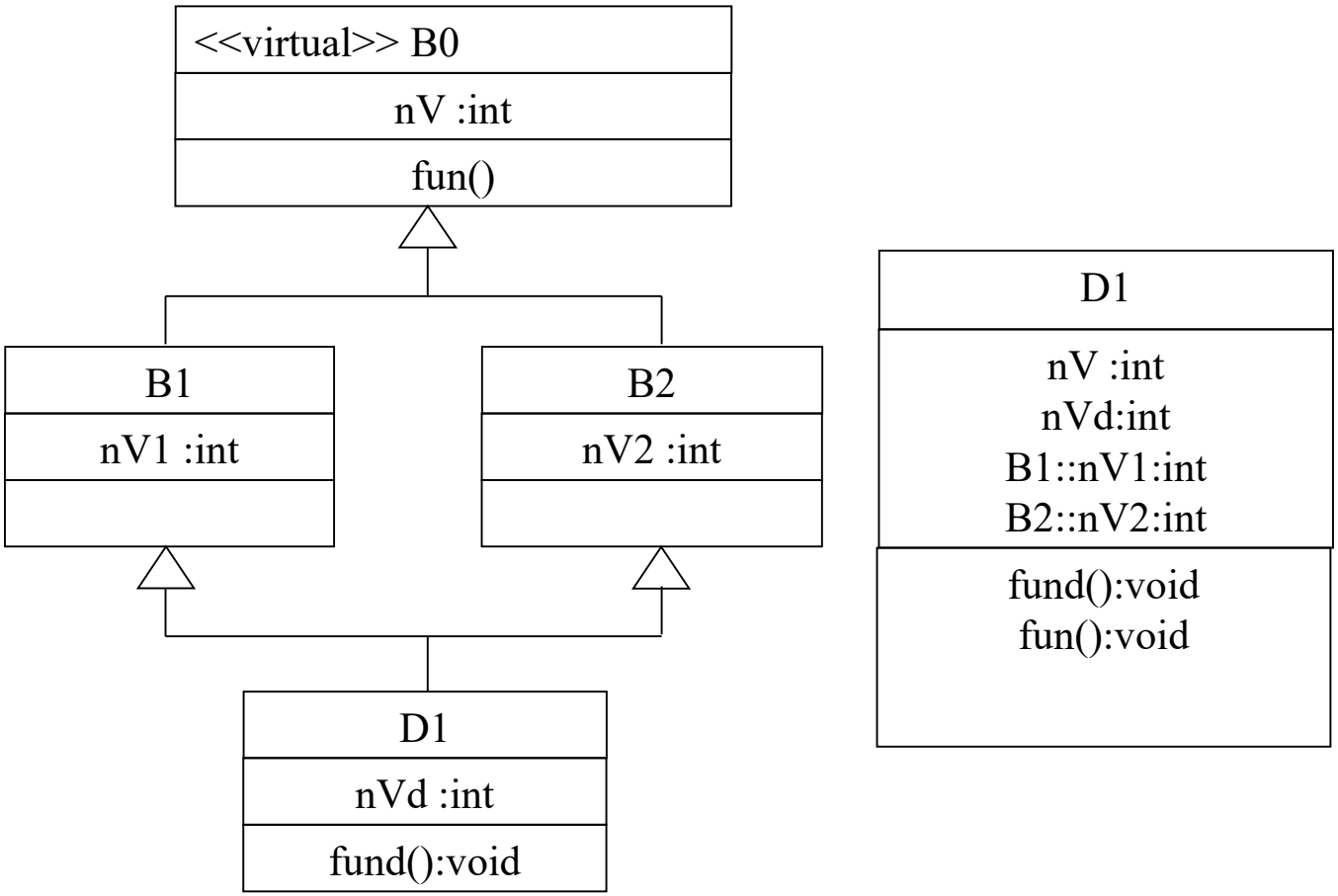
下面的访问是正确的：

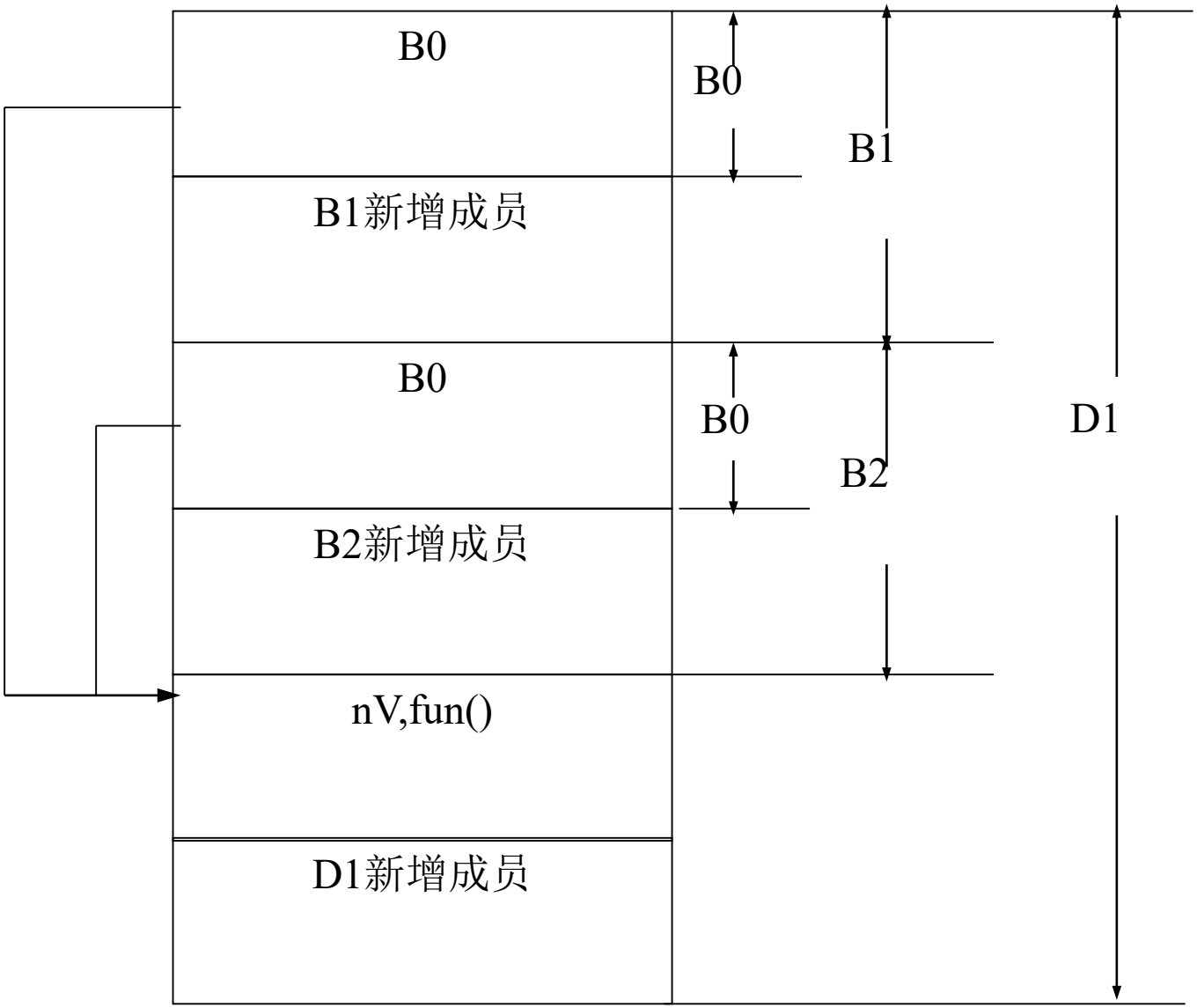
```
C  cobj;
cobj.b;
```

虚基类的派生类对象存储结构示意图



例7-8 虚基类举例


<<virtual>> B0
nV :int
fun()
B1
nV1 :int
B2
nV2 :int
D1
nVd :int
fund():void
D1
nV :int
nVd:int
B1::nV1:int
B2::nV2:int
fund():void
fun():void


B0
B0
B1
B1新增成员
B0
B0
B2
D1
B2新增成员
nV,fun()
D1新增成员

```
#include <iostream>
using namecpace std;
class B0 //声明基类B0
{ public: //外部接口
      int nV;
      void fun(){cout<<"Member of B0"<<endl;}
};
class B1: virtual public B0  //B0为虚基类，派生B1类
{ public: //新增外部接口
      int nV1;
};
class B2: virtual public B0  //B0为虚基类，派生B2类
{  public:      //新增外部接口
      int nV2;
};
class D1: public B1, public B2 //派生类D1声明
{  public:      //新增外部接口
      int nVd;
      void fund(){cout<<"Member of D1"<<endl;}
};
void main()  //程序主函数
{   D1 d1;     //声明D1类对象d1
      d1.nV=2;     //使用最远基类成员
      d1.fun();
}
```

虚基类及其派生类构造函数

- 建立对象时所指定的类称为最远派生类。
- 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。
- 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的缺省构造函数。
- 在建立对象时，只有最远派生类的构造函数调用虚基类的构造函数，该派生类的其它基类对虚基类构造函数的调用被忽略。

有虚基类时的构造函数举例

```
#include <iostream>
using namespace std;
class B0  //声明基类B0
{ public:  //外部接口
      B0(int n){ nV=n; cout<<"Constructor of B0"<<endl;}
      int nV;
      void fun(){cout<<"Member of B0"<<endl;}
};
class B1: virtual public B0
{  public:
      B1(int a) : B0(a) {} // 虚基类的构造函数在调用时被忽略，但必须声明
      int nV1;
};
class B2: virtual public B0
{  public:
      B2(int a) : B0(a) {} //虚基类的构造函数在调用时被忽略，但必须声明
      int nV2;
};
class D1: public B1, public B2
{public:
      D1(int a) : B0(a), B1(a), B2(a) {}
      int nVd;
      void fund(){cout<<"Member of D1"<<endl;}
};
void main() {
      D1 d1(1);
      d1.nV=2;
      d1.fun();
}
```

哪个结果正确?

运行结果1：
Constructor of B0
Constructor of B0
Constructor of B0
Member of B0

运行结果2：
Constructor of B0
Member of B0

课后作业

- 完成课后题: 7-1, 7-4, 7-10, 7-11, 7-12
- 截止时间：2022-4-29 晚上9点
- 作业载体：Word文档（可以粘贴手写作业图片，对于需要运行程序的题目，请提供结果截图）
- 提交方式：各班班长按学号和姓名收齐，备注好班级发到邮箱ruim@jlu.edu.cn

课程纲要

① 第七章

类的继承与派生

类成员的访问控制

单继承与多继承

类型兼容性规则

构造函数、析构函数

类成员的标识与访问

Q&A