2019年春

程序设计实习(I): C++程序设计

第五讲 继 承

刘家瑛

liujiaying@pku.edu.cn

上节知识点回顾

- 内联函数 / 内联成员函数(2种方式) **inline**关键字
- 函数重载 / 缺省值
- **静态**成员变量和静态成员函数 **static**
  - 所有对象共享/**没有对象生成**也能直接访问 / 本质是**全局变量**
  - 静态成员函数中, 不能访问非静态成员变量 / 不能调用非静态成员函数
- **const**
  - 不能通过const指针修改其指向的地方的内容
  - 不能把常量指针赋值给非常量指针
  - 常量对象 / 常量成员函数重载
- **成员对象和封闭类**
  - 封闭类的构造函数时, 添加**初始化列表**
- **友元:** 友元函数和友元类

友元 (friends)

- 友元分为友元函数和友元类两种
  - 友元函数: 一个类的友元函数可以访问该类的私有成员

```
class CCar ; //提前声明 CCar类, 以便后面的CDriver类使用
class CDriver{
    public:
        void ModifyCar( CCar * pCar) ; //改装汽车
};
class CCar{
    private:
        int price;
    friend int MostExpensiveCar( CCar cars[],  int total); //声明友元
    friend void CDriver::ModifyCar(CCar * pCar);        //声明友元
};
```

```
void CDriver::ModifyCar( CCar * pCar) {
        pCar->price += 1000; //汽车改装后价值增加
}
int MostExpensiveCar( CCar cars[],  int total) {
//求最贵汽车的价格
        int tmpMax = -1;
        for( int i = 0; i < total; ++i )
                if( cars[i].price > tmpMax)
                        tmpMax = cars[i].price;
        return tmpMax;
}
int main()
{
        return 0;
}
```


- 可以将一个类的成员函数(包括构造，析构函数)说明为另一个类的友元

```
class B {
    public:
        void function();
};

class A {
        friend void B::function();
};
```


2. 友元类: 如果A是B的友元类, 那么A的成员函数可以访问B的私有成员

```
class CCar{
    private:
        int price;
        friend class CDriver;    //声明CDriver为友元类
};
class CDriver{
    public:
        CCar myCar;
        void ModifyCar(){  //改装汽车
            myCar.price += 1000;  //因CDriver是CCar的友元类,
        }                          //故此处可以访问其私有成员
};
int main(){    return 0;    }
```

- 友元类之间的关系不能传递, 不能继承

this 指针

- C++程序到C程序的翻译:

```
class CCar{
    public:
        int price;
        void SetPrice(int p);
};
void CCar::SetPrice(int p){
    price = p;
}
int main(){
    CCar car;
    car.SetPrice(20000);
    return 0;
}
```

```
struct CCar{
    int price;
};
void SetPrice(CCar * this,  int p){
    this->price = p;
}
int main(){
    struct CCar car;
    SetPrice(& car,  20000);
    return 0;
}
```


this 指针

- 非静态成员函数中可以直接使用 this 来代表指向该函数作用的对象的指针

```
class Complex {
    public:
        double real,  imag;
        Complex(double r,  double i):real(r),  imag(i) { }
        Complex AddOne() {
            this->real ++; //this指针类型是Complex *
            return * this;
        }
};
int main()  {
        Complex c1(1, 1), c2(0, 0);
        c2 = c1.AddOne();
        cout << c2.real << ", " << c2.imag << endl; //输出 2, 1
        return 0;
}
```

上节课知识点复习

- 下面说法哪个不正确?

A) 静态成员函数内部不能访问同类的非静态成员变量，也不能调用同类的非静态成员函数

B) 非静态成员函数不能访问静态成员变量

C) 静态成员变量被所有对象所共享

D) 在没有任何对象存在的情况下，也可以访问类的静态成员

上节课知识点复习

- 以下关于友元的说法哪个是不正确的?

A) 一个类的友元函数中可以访问该类对象的私有成员

B) 友元类关系是相互的, 即若类A是类B的友元, 则类B也是类A的友元

C) 在一个类中可以将另一个类的成员函数声明为友元

D) 类之间的友元关系不能传递

上节课知识点复习

- 以下说法正确的是：

A) 成员对象都是用无参构造函数初始化的

B) 封闭类中成员对象的构造函数先于封闭类的构造函数被调用

C) 封闭类中成员对象的析构函数先于封闭类的析构函数被调用

D) 若封闭类有多个成员对象，则它们的初始化顺序取决于封闭类构造函数中的成员初始化列表

上节课知识点复习

- 以下说法不正确的是：

A) 静态成员函数中不能使用this指针

B) this指针就是指向成员函数所作用的对象的指针

C) 每个对象的空间中都存放着一个this指针

D) 类的非静态成员函数，真实的参数比所写的参数多1

上节内容回顾

- 三种运算符重载的实现方式
  - 重载为普通函数
  - 重载为成员函数
  - 重载为友元
- **常见的各种运算符重载**
  - 流运算符(>>/<<)
  - 自增/自减运算符(++/--)
  - 赋值号(=)
  - 下标运算符([])
  - 类型转换运算符

上节课知识点复习

- 关于运算符重载，下列表达中正确的是 _____.

A) C++已有的任何运算符都可以重载

B) 运算符函数的返回类型不能声明为基本数据类型

C) 在类中，一个运算符可以对应多个不同的重载函数

D) 可以通过运算符重载来创建C++中原来没有的运算符

上节课知识点复习

- 重载 "<<" 用于将自定义的对象通过cout输出时, 以下说法哪个是正确的?

A) 可以将 "<<" 重载为ostream 类的成员函数, 返回值类型是 ostream &

B) 可以将 "<<" 重载为全局函数, 第一个参数以及返回值, 类型都是 ostream

C) 可以将 "<<" 重载为全局函数, 第一个参数以及返回值, 类型都是 ostream &

D) 可以将 "<<" 重载为 ostream 类的成员函数, 返回值类型是 ostream

上节课知识点复习

- 以下关于赋值运算符重载的说法，正确的是：

A) 赋值运算符重载成全局函数时，应该有两个参数

B) 赋值运算符重载的唯一目的就是使得可以用其他类型的变量或常量给对象复制

C) 赋值运算符重载时，返回值设为其所作用的对象的引用，是符合赋值运算符使用习惯的做法

D) 赋值运算符重载时，返回值设为void是符合赋值运算符使用习惯的做法

上节课知识点复习

- 如果将 [ ] 运算符重载成一个类的成员函数，则该重载函数有几个参数?

A) 0

B) 1

C) 2

D) 3

上节课知识点复习

- 如果将运算符 "*" 重载为某个类的成员运算符 (也即成员函数), 则该成员函数的参数个数是:

A) 0个

B) 1个

C) 2个

D) 0个1个均可

上节课知识点复习

- 如何区分自增运算符重载的前置形式和后置形式?

A) 重载时, 前置形式的函数名是 ++ operator, 后置形式的函数名是 operator ++

B) 后置形式比前置形式多一个 int 类型的参数

C) 无法区分, 使用时不管前置形式还是后置形式, 都调用相同的重载函数

D) 前置形式比后置形式多了一个int类型的参数

上节课知识点复习

程序输出结果如下，请填空

0

5

```
class A {
public:
    int val;
        A(_____ ){   val = n;   }
        _____ GetObj() {
            return _____;
        }
};
```

```
int main(){
    A a;    cout <<a.val << endl;
    a.GetObj() = 5;
    cout << a.val << endl;
    return 0;
}
```


上节课知识点复习

程序输出结果如下，请填空

0

5

```
class A {
public:
    int val;
    A(_____ ){   val = n;   }
    _____ GetObj() {
        return _____;
    }
};
```

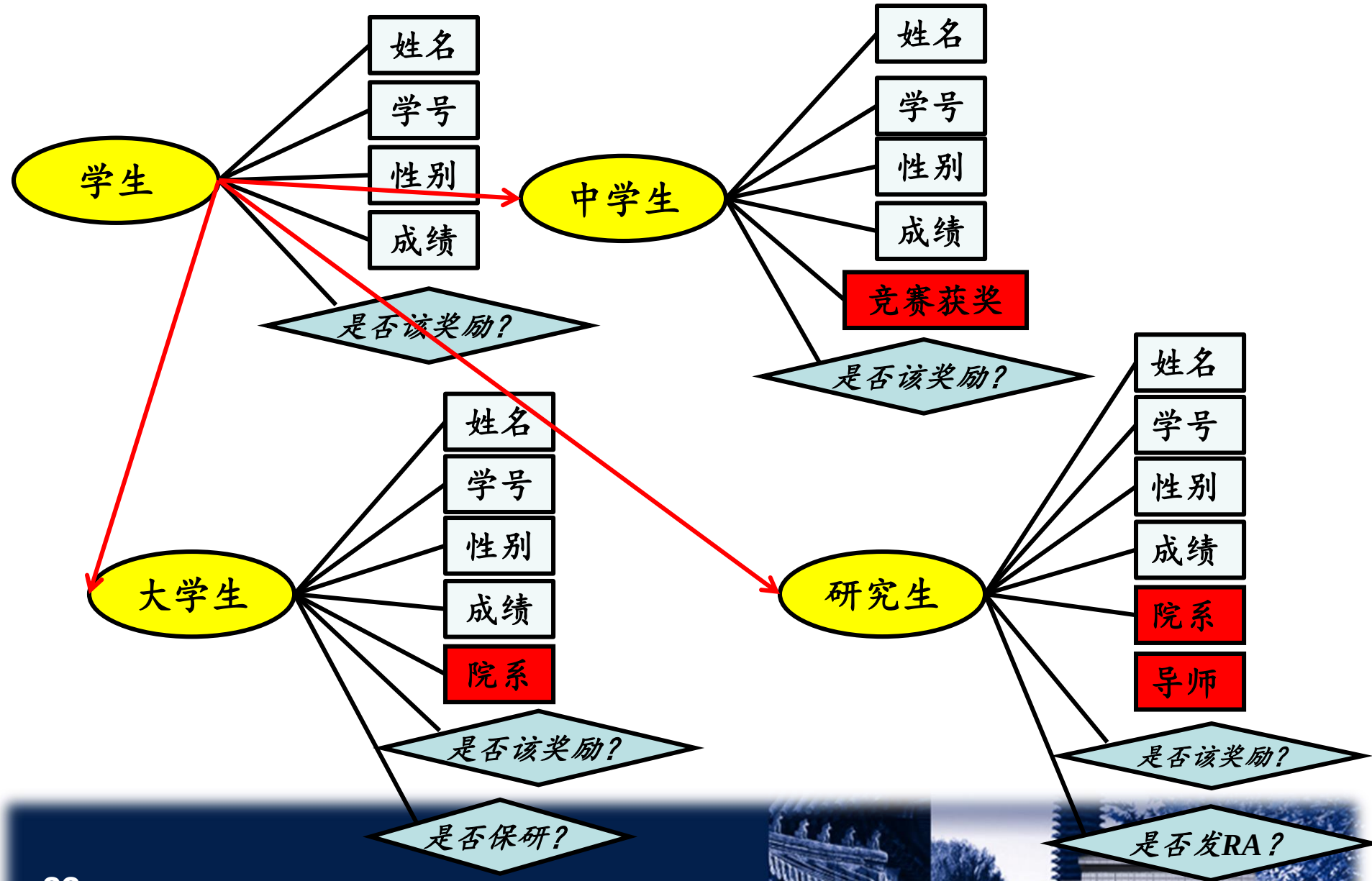
```
int main(){
    A a;    cout <<a.val << endl;
    a.GetObj() = 5;
    cout << a.val << endl;
    return 0;
}
```

```
A(int n=0)
A& GetObj(){
        return *this;
}
或者
int& GetObj(){
        return val;
}
```

主要内容

- 基本概念: 继承, 基类, 派生类
- 派生类的成员组成, 可见性
- 派生类的构造, 析构
- 派生类与基类的指针类型转换

继承机制：从例子开始


学生
姓名
学号
性别
成绩
是否该奖励?
中学生
姓名
学号
性别
成绩
竞赛获奖
是否该奖励?
大学生
姓名
学号
性别
成绩
院系
是否该奖励?
是否保研?
研究生
姓名
学号
性别
成绩
院系
导师
是否该奖励?
是否发RA?

继承与派生的概念

继承

在定义一个新的类B时，如果该类与某个已有的类A相似

(指的是B拥有A的全部特点)

A—**基类(父类)**，B—基类的**派生类(子类)**

- **派生类**是通过对基类进行**修改**和**扩充**得到的
- 扩充：在派生类中，可以**添加**新的成员变量和成员函数
- 修改：在派生类中，可以**重新编写**从基类继承得到的成员
- 派生类一经定义后，可以独立使用，不依赖于基类

继承与派生的概念

继承

C++中，从一个类派生出另一个类的写法：

class 派生类名：派生方式说明符 基类名

{

…

}

- 派生类拥有基类的**全部成员**，包括
  - private/protected/public成员变量
  - private/protected/public方法
- 在派生类的各个成员函数中，**不能访问基类中的private成员**

需要继承机制的例子

- 所有的学生都有一些**共同属性和方法**
  - **属性:** 姓名, 学号, 性别, 成绩等
  - **方法:** 判断是否该留级, 判断是否该奖励等
- 不同的学生, 比如中学生, 大学生, 研究生, 又有各自**不同的属性和方法**

e.g.

- 大学生有系的属性, 而中学生没有
- 研究生有导师的属性
- 中学生竞赛、特长加分之类的属性

需要继承机制的例子

- 如果为每类学生都编写一个类

  →不少重复的代码，浪费

- **比较好的做法：**
  - 编写一个"学生"类，概括了各种学生的共同特点
  - 从"学生"类派生出"大学生"类，"中学生"类，"研究生类"

```
class CStudent {
    private:
        char szName[20];
        int nAge;
        int nSex;

    public:
        bool IsThreeGood() { };
        int SetSex( int nSex_ ) { nSex = nSex_ ; }
        void SetName( char * szName_ ) { strcpy( szName,
                                                 szName_ ); }
        // ...
};
```


```
//派生类的写法是:
//      类名: public 基类名
class CUndergraduateStudent : public CStudent {
    private:
        int nDepartment;
    public:
        bool IsThreeGood() {  ...  };  //覆盖
        bool CanBaoYan() { .... };
};
```


```
class CGraduateStudent : public CStudent {
        private:
                int nDepartment;
                char szSupervisorName[20];
        public:
                int CountSalary() { ... };

};
```


复合与继承

- **继承:** "**是**" 关系
  - 基类 A
  - B是基类A的派生类
  - 逻辑上要求："一个B对象也**是**一个A对象"
- **复合:** "**有**" 关系
  - 复合,即一个类的对象拥有作为其成员的其它类的对象
  - d是类C的成员
  - d是类D的一个对象
  - 复合关系满足：C类中 "**有**" 成员对象d

继承的使用

- 写了一个 CMan 类代表男人

→又发现需要一个CWoman类来代表女人

  - 仅仅因为CWoman类和CMan类有共同之处, 就让CWoman类从CMan类派生而来, 是不合理的
  - 因为 "一个女人也是一个男人" 从逻辑上不成立

- 好的做法是概括男人和女人共同特点, 写一个 CHuman类, 代表 "人", 然后CMan和CWoman都从CHuman派生

复合关系的使用

- 几何形体程序中，需要写"点"类，也需要写"圆"类，两者的关系就是复合关系
- 每一个"圆"对象里都包含(有)一个"点"对象，这个"点"对象就是圆心

```
class CPoint{
    double x,  y;
    friend class CCircle;  //便于CCirle类操作其圆心
};
class CCircle{
     double r;
     CPoint center;
};
```

复合关系的使用

写一个小区养狗管理程序

- 需要写一个"业主"类，还需要写一个"狗"类
- 而狗是有"主人"的，主人当然是业主

(假定狗只有一个主人，但一个业主可以有最多10条狗)

复合关系的使用

```
class CDog;
class CMaster
{
    CDog dogs[10];
};
class CDog
{
    CMaster m;
};
```

这段程序有问题吗?

复合关系的使用

```
class CDog;
class CMaster
{
    CDog dogs[10]; //无法编译, Cdog对象无定义
};
class CDog
{
    CMaster m;
};
```

避免循环定义的方式: 在一个类中使用另一个类的指针, 而不是对象作为成员变量

复合关系的使用

正确写法：

- 为"狗"类设一个"业主"类的成员对象指针
- 为"业主"类设一个"狗"类的对象指针数组

```
class CDog;
class CMaster
{
    CDog * dogs[10];
};
class CDog
{
    CMaster * m;
};
```


继 承

- 派生类可以定义一个**和基类成员同名**的成员，这叫**覆盖**
- 在派生类中访问这类成员时，缺省的情况是访问派生类中定义的成员
- 要在派生类中访问由**基类定义的同名成员**时，要使用**作用域符号::**

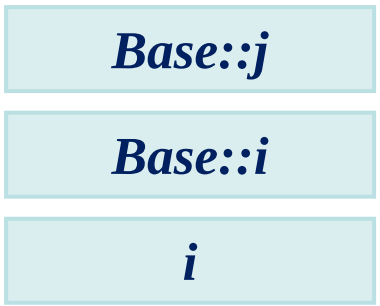
基类和派生类有同名成员的情况

```
class base {
        int j;
public:
        int i;
        void func();
};
```

```
class derived : public base{
public:
        int i;
        void access();
        void func();
};
```

```
void derived::access(){
        j = 5;          //error
        i = 5;          //派生类的 i
        base::i = 5;    //基类的 i
        func();         //派生类的
        base::func();   //基类的
}
derived  obj;
obj.i = 1;
obj.base::i = 1;
```

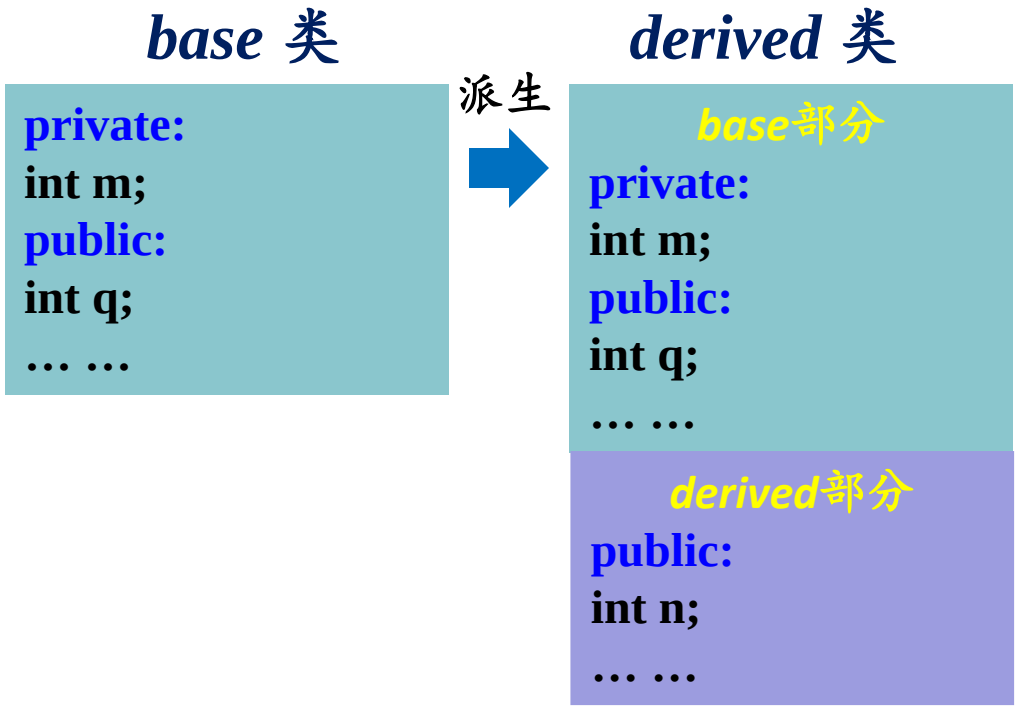
Obj占用的存储空间



一般，基类和派生类不定义同名成员变量

public继承

```
class base {
        private:
            int m;
        public:
            int q;
            base(int i=0):m(i){}
};
class derived : public base {
        public:
            int n;
            derived(int j=0):base(j){}
};
base b;
derived d;
```



另一种存取权限说明符:protected

- 基类的**private成员**：可以被下列函数访问
  - 基类的成员函数
  - 基类的友员函数
- 基类的**public成员**：可以被下列函数访问
  - 基类的成员函数
  - 基类的友员函数
  - 派生类的成员函数
  - 派生类的友员函数
  - 其他的函数

另一种存取权限说明符:protected

- 基类的**protected成员**: 可以被下列函数访问
- 可访问范围比private成员大, 比public成员小
  - 基类的成员函数
  - 基类的友员函数
  - **派生类的成员函数**可以访问**当前对象**的基类的保护成员

→ this指针指向的对象

保护成员

```
class  Father {
      private: int nPrivate;          //私有成员
      public:   int nPublic;          //公有成员
      protected: int nProtected;   // 保护成员
};
class Son : public Father{
      void AccessFather () {
            nPublic = 1; // ok;
            nPrivate = 1;  // wrong
            nProtected = 1; // OK, 访问从基类继承的protected成员
            Father f;
            f.nProtected = 1; //wrong, f不是函数所作用的当前对象
      }
};
```

```
int main()
{
        Father f;
        Son s;
        f.nPublic = 1;        // OK
        s.nPublic = 1;        // OK
        f.nProtected = 1;     // error
        f.nPrivate = 1;       // error
        s.nProtected = 1;     // error
        s.nPrivate = 1;       // error
        return 0;
}
```


派生类的构造函数

```
class Bug  {
        private :
                int nLegs;
                int nColor;
        public:
                int nType;
                Bug (int legs,  int color);
                void PrintBug(){ };
};
class FlyBug: public Bug // FlyBug是Bug的派生类
{
                int nWings;
        public:
                FlyBug(int legs,  int color,  int wings);
};
```


```
Bug::Bug(int legs,  int color){
        nLegs = legs;
        nColor = color;
}
//错误的FlyBug构造函数
FlyBug::FlyBug (int legs,  int color,  int wings){
        nLegs = legs;     // 不能访问
        nColor = color;  // 不能访问
        nType = 1; // ok
        nWings = wings;
}
//正确的FlyBug构造函数
FlyBug::FlyBug (int legs,  int color,  int wings):Bug( legs,  color){
        nWings = wings;
}
```


```
int main() {
    FlyBug fb (2, 3, 4);
    fb.PrintBug();
    fb.nType = 1;
    fb.nLegs = 2 ;  // error!  nLegs is private
    return 0;
}
```


FlyBug fb (2, 3, 4);

- 在创建**派生类的对象**时，需要调用**基类的构造函数**
  - 初始化派生类对象中从基类继承的成员
  - 在执行**派生类的构造函数之前**，总是先执行**基类的构造函数**
- 调用基类构造函数的两种方式
  - **显式方式:** 在派生类的构造函数中，为基类构造函数提供参数

    derived::derived(arg_derived-list):base(arg_base-list)
  - **隐式方式:** 在派生类的构造函数中，省略基类构造函数时，派生类的构造函数则自动调用基类的默认构造函数
- **派生类的析构函数**被执行时，执行完派生类的析构函数后，自动调用**基类的析构函数**

```
class Base {
        public:
                int n;
                Base(int i):n(i)
                { cout << "Base " << n << " constructed" << endl; }
                ~Base()
                { cout << "Base " << n << " destructed" << endl;}
};
class Derived:public Base  {
        public:
                Derived(int i):Base(i)
                {  cout << "Derived constructed" << endl;  }
                ~Derived()
                {  cout << "Derived destructed" << endl;  }

};
```


```
int main()
{
    Derived Obj(3);
    return 0;
}
```

输出结果:

```
Base 3 constructed
Derived constructed
Derived destructed
Base 3 destructed
```


包含成员对象的派生类的构造函数

```
class Skill{
    public:
        Skill(int n) { }
};
class FlyBug: public Bug {
        int nWings;
        Skill  sk1,  sk2;
   public:
        FlyBug( int legs,  int color,  int wings);
};
FlyBug::FlyBug( int legs,  int color,  int wings):
        Bug(legs,  color),  sk1(5),  sk2(color)  {
        nWings = wings;
}
```

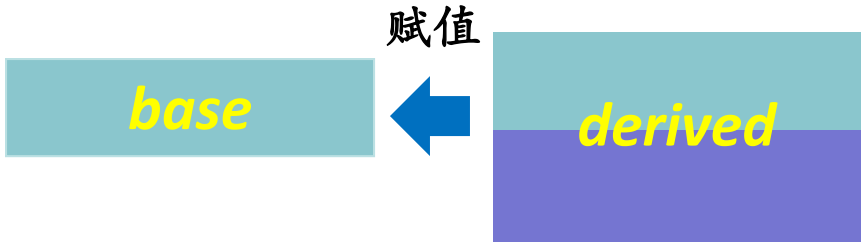

- 创建派生类的对象时，执行一个派生类的构造函数之前：
  - 调用**基类**的构造函数

    初始化派生类对象中从基类继承的成员
  - 调用**成员对象类**的构造函数

    初始化派生类对象中成员对象
- 派生类析构函数被执行时，执行完派生类的析构函数后：
  - 调用**成员对象类**的析构函数
  - 调用**基类**的析构函数
- **析构函数的调用顺序与构造函数的调用顺序相反**

public继承的赋值兼容规则

```
class base { };
class derived : public base { };
base b;
derived d;
```



- 派生类的对象可以赋值给基类对象

  b = d;

- 派生类对象可以初始化基类引用

  base & br = d;

- 派生类对象的地址可以赋值给基类指针

  base * pb = & d;

- 如果派生方式是 private或protected, 则上述三条不可行

该对象不包含派生类定义的成员，因为没有派生类成员的存储空间 (派生类部分在赋值时被 “切掉”) *C++ Primer P489

基类与派生类的指针强制转换

- 公有派生的情况下，
- 派生类对象的地址可以直接赋值给基类指针

  Base * ptrBase = &objDerived;

  - ptrBase指向的是一个Derived类的对象
  - *ptrBase可以看作一个Base类的对象，访问它的public成员
  - 直接通过ptrBase，不能够访问**objDerived由Derived类扩展的成员**
- 即便基类指针指向的是一个派生类的对象，也不能通过基类指针访问基类没有，而派生类中有的成员

基类与派生类的指针强制转换

- 通过强制指针类型转换, 可以把ptrBase转换成Derived类的指针

  Base * ptrBase = &objDerived;

  Derived *ptrDerived = (Derived *) ptrBase;

  - 程序员要保证ptrBase指向的是一个Derived类的对象,

  否则很容易会出错

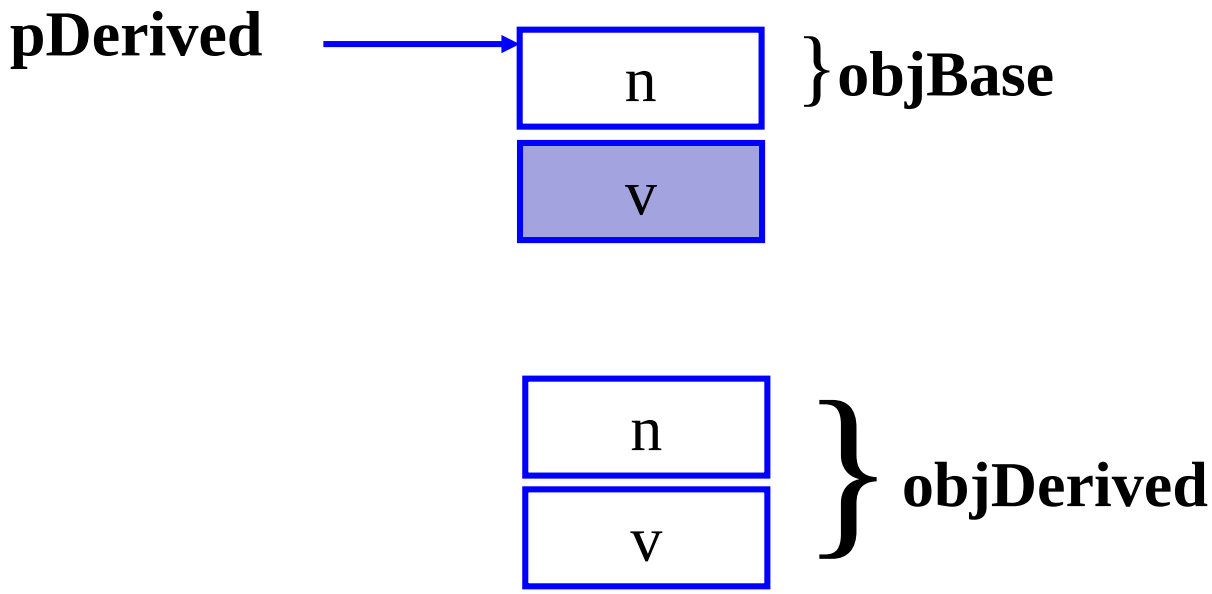
```
#include <iostream>
using namespace std;
class Base {
        protected:
                int n;
        public:
                Base(int i):n(i){
                        cout << "Base " << n <<
                                " constructed" << endl;    }
                ~Base() {
                        cout << "Base " << n <<
                                " destructed" << endl;
                }
                void Print() { cout << "Base:n=" << n << endl;}
};
```

```
class Derived:public Base  {
        public:
                int v;
                Derived(int i):Base(i),  v(2 * i) {
                    cout << "Derived constructed" << endl;
                }
                ~Derived()   {
                    cout << "Derived destructed" << endl;
                }
                void Func() { } ;
                void Print() {
                    cout << "Derived:v=" << v << endl;
                    cout << "Derived:n=" << n << endl;
                }
};
```

```
int main()  {
    Base objBase(5);
    Derived objDerived(3);
    Base * pBase = & objDerived ;
    //pBase->Func();    //err; Base类没有Func()成员函数
    //pBase->v = 5;     //err; Base类没有v成员变量
    pBase->Print();     //调用基类函数Print()
    //Derived * pDerived = & objBase;    //error
    Derived * pDerived = (Derived *)(& objBase);
    pDerived->Print();   //慎用，可能出现不可预期的错误
    pDerived->v = 128;   //往别人的空间里写入数据
                         //→可能引起问题
    objDerived.Print();
    return 0;
}
```

根据指针/引用类型
来决定调用的函数

Derived * pDerived = (Derived *)(& objBase);



输出结果:

```
Base 5 constructed
Base 3 constructed
Derived constructed
Base:n=3
Derived:v=0
Derived:n=5
Derived:v=6
Derived:n=3
Derived destructed
Base 3 destructed
Base 5 destructed
```


直接基类与间接基类

- 类A派生类B, 类B可再派生类C, 类C派生类D, ……
  - 类A是类B的直接基类
  - 类B是类C的直接基类, 类A是类C的间接基类
  - 类C是类D的直接基类, 类A, B是类D的间接基类
- 在声明派生类时, 派生类的首部只需要列出它的**直接基类**
  - 派生类的首部**不要列出它的间接基类**
  - 派生类沿着类的层次自动向上继承它的间接基类
  - 派生类的成员包括
    - 派生类自己定义的成员
    - 直接基类中定义的成员
    - 间接基类的全部成员

```
#include <iostream>
using namespace std;
class Base {
    public:
        int n;
        Base(int i):n(i)  {
            cout << "Base " << n << " constructed" << endl;
         }
         ~Base() {
             cout << "Base " << n << " destructed" << endl;
         }

}; //基类
```


```
class Derived: public Base
{
    public:
        Derived(int i):Base(i) {
            cout << "Derived constructed" << endl;
        }
        ~Derived() {
            cout << "Derived destructed" << endl;
        }

};
```


```
class MoreDerived: public Derived {
    public:
        MoreDerived():Derived(4) {
            cout << "More Derived constructed" << endl;
        }
        ~MoreDerived() {
            cout << "More Derived destructed" << endl;
        }
};

int main()
{
    MoreDerived Obj;
    return 0;
}
```


输出结果:

Base 4 constructed
Derived constructed
More Derived constructed
More Derived destructed
Derived destructed
Base 4 destructed

多继承（考试不要求）

- 一个类可以从多个基类派生而来，以继承多个基类的成员—这种派生称作"**多重继承**"

class derived:access-specifier$_1$ base$_1$, access-specifier$_2$ base$_2$, … {

……

};

access-specifier$_i$可以是private, protected, public之一

多继承的派生类构造函数

```
class base1 {
    int i;
public:
    base1(int n) { i = n; }
};
class base2 {
    int j;
public:
    base2(int n) { j = n; }
};
class  derived : public base1, public base2{
    public:
        derived( int x );
};
derived::derived( int x ): base1(x), base2(0) { }
```

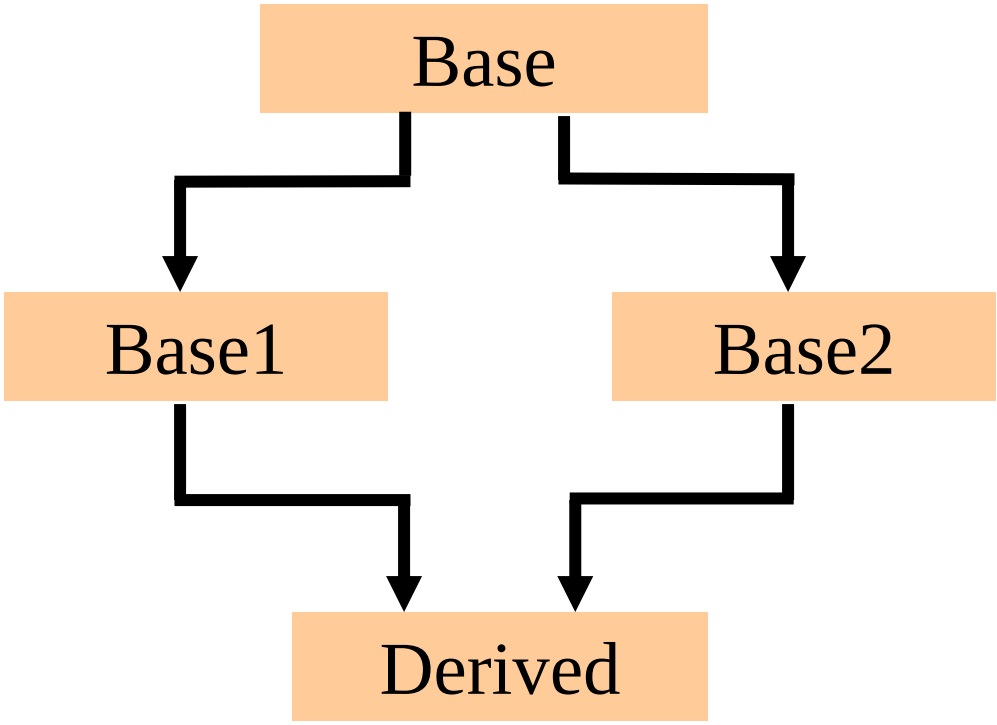
多继承的派生类构造函数

- 多重继承中，派生类对象创建时，先按继承顺序调用基类的构造函数，然后再调用派生类的构造函数
- 如果派生类是封闭类，那么成员对象的构造函数在基类的构造函数调用结束后依次调用，最后才调用派生类的构造函数
- 多重继承中，派生类对象的创建过程
  - 按继承顺序调用基类的构造函数
  - 依次调用成员对象的构造函数
  - 调用派生类的构造函数

多继承中基类的构造函数调用

```
class Base {
public:
        int val;
        Base() { cout << "Base Constructor" << endl; }
        ~Base() {
                cout << "Base Destructor" << endl;
        }
};
class Base1:public Base {  };
class Base2:public Base {  };
class Derived:public Base1, public Base2 {   };
```


```
int main() {
        Derived d;
}
```



输出结果:

Base Constructor
Base Constructor
Base Destructor
Base Destructor

基类Base的构造函数和析构函数都被调用两次

多重继承的二义性

```
class base1 {
    private:
        int b1;
        void set( int i) {  b1 = 1; }
    public : int i;
};
class base2 {
    private:
        int b2;
    public:
        void set( int i)  { b2 = i ;}
        int get() {  return b2 ; }
        int i;
};
```


```
class derived :public base1, public base2{
    public:
        void  print()  {
              printf( “%d”, get() );
              set(5);  //二义性, error
              base2::set(5) ; // ok
              base1::set(5);  // error, set is private in base1
        }
};
```

derived对象的属性

int base1:: private b1
int base1:: public i
int base2:: private b2
int base2:: public i

derived对象的操作

void base1:: private set(int i)
void base2:: public set(int i)
int base2:: public get(int i)
public void print()

```
int main ()  {
      derived d;
      d.set(10);     //二义性
      d.base1::set(10);  // error, can't access private member
      d.base2::set(5);
      d.base1::i =5;
      d.base2::i = 5;
      Return 0;
}
```

二义性检查在访问权限检查之前进行

不能靠成员的访问权限来消除二义性

```
class A { public: void fun() ; }
class B { private: void fun(); }
class C : public A, public B { } ;
C obj;
Obj.fun() ; // 二义性
```


总　结

- 基本概念: 继承, 基类, 派生类
  - 合理派生
- 派生类的成员组成, 可见性
  - private/protected成员的继承性
  - 派生类的成员函数—**不能访问基类中的private成员**
  - 派生类的成员函数—访问**当前对象**的**基类的protected成员**
- 派生类的构造, 析构
  - 构造顺序：基类, 对象成员, 派生
  - 析构反之
- 派生类与基类的指针类型转换
  - f(派生类)→y(基类对象): **f: 对象/对象地址; y: 对象/引用/指针**
  - 基类指针→(强制指针类型转换)派生类指针