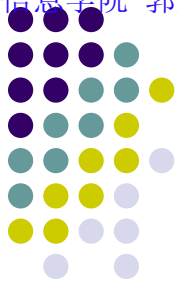
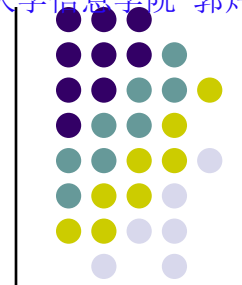


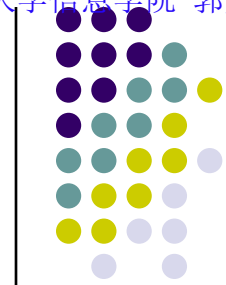
新标准C++程序设计

北京大学信息学院　郭 炜

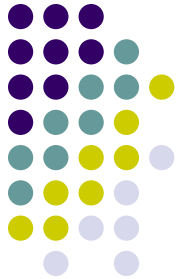
GWPL@PKU.EDU.CN



继承



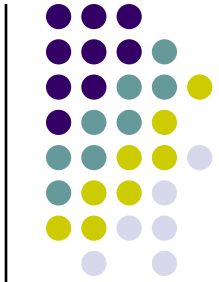
继承与派生的概念



- 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)，那么就可以把A作为一个基类，而把B作为基类的一个派生类(也称子类)。
  - 派生类是通过对基类进行修改和扩充得到的。在派生类中，可以扩充新的成员变量和成员函数。
  - 派生类一经定义后，可以独立使用，不依赖于基类。
- 派生类拥有基类的全部成员，包括：(private、protected、public)属性和(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。

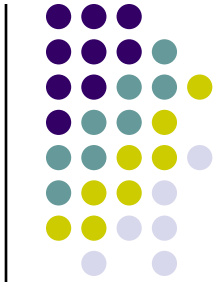
需要继承机制的例子：

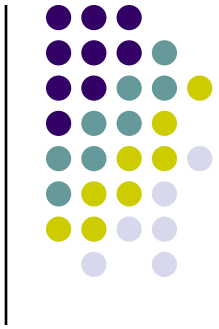
- 所有的学生都有一些共同属性和方法，比如姓名，学号，性别，成绩等属性，判断是否该留级，判断是否该奖励之类的方法。
- 而不同的学生，比如中学生，大学生，研究生，又有各自不同的属性和方法，比如大学生有系的属性，而中学生没有，研究生有导师的属性，中学生竞赛、特长加分之类的属性。
- 如果为每类学生都编写一个类，显然会有不少重复的代码，浪费。
- 比较好的做法是编写一个“学生”类，概括了各种学生的共同特点，然后从“学生”类派生出“大学生”类，“中学生”类，“研究生类”。



```
class CStudent {
        private:
                string sName;
                int nAge;
        public:
        bool IsThreeGood() { };
        void SetName( const string & name )
        { sName = name; }
        //.....
};
class CUndergraduateStudent: public CStudent {
        private:
                int nDepartment;
        public:
                bool IsThreeGood() {  .....  }; //覆盖
                bool CanBaoYan() { .... };
}; // 派生类的写法是：类名: public 基类名
```

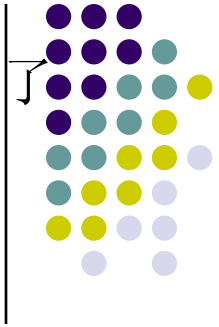
```
class CGraduatedStudent:public CStudent {
        private:
                int nDepartment;
                char szMentorName[20];
        public:
                int CountSalary() { ... };
};
```

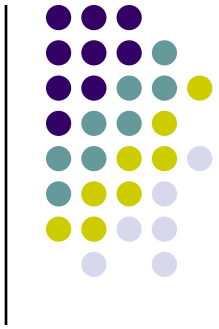




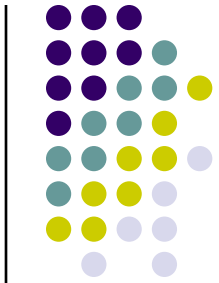
```
#include <iostream>
#include <string>
using namespace std;
class CStudent
{
    private:
            string name;
            string id;  //学号
            char gender; //性别,'F'代表女,'M'代表男
            int age;
      public:
            void PrintInfo();
            void SetInfo( const string & name_,const
string & id_, int age_,char gender_ );
          string GetName() {  return name;  }
};
```

```
class CUndergraduateStudent:public CStudent  //本科生类，继承了CStudent类
{
        private:
                string department; //学生所属的系的名称
        public:
                void QulifiedForBaoyan() {//给予保研资格
                        cout << "qulified for baoyan" << endl;
                }
                void PrintInfo() {
                        CStudent::PrintInfo(); //调用基类的PrintInfo
                        cout << "Department:" << department <<endl;
                }
                void SetInfo( const string & name_,const string & id_,
                        int age_,char gender_ ,const string &
                        department_) {
                        CStudent::SetInfo(name_,id_,age_,gender_);
                        //调用基类的SetInfo
                        department = department_;
                }
};
```

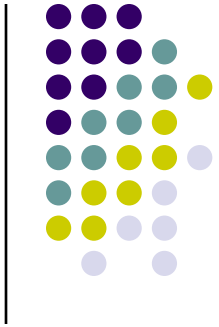




```
void CStudent::PrintInfo()
{
	cout << "Name:" << name << endl;
	cout << "ID:" << id << endl;
	cout << "Age:" << age << endl;
	cout << "Gender:" << gender << endl;
}
void CStudent::SetInfo( const string & name_,const string &
id_, int age_,char gender_ )
{
	name = name_;
	id = id_;
	age = age_;
	gender = gender_;
}
```

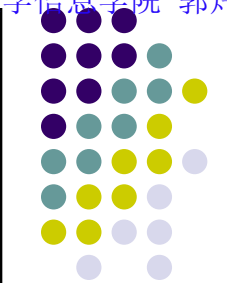


```
int main()
{
      CStudent s1;
      CUndergraduateStudent s2;
      s2.SetInfo("Harry Potter",
         "118829212",19,'M',"Computer Science");
      cout <<  s2.GetName() << " " ;
      s2.QulifiedForBaoyan ();
      s2.PrintInfo ();
      cout << "sizeof(string)=" << sizeof(string) << endl;
      cout << "sizeof(CStudent)=" << sizeof(CStudent) <<
endl;
    cout << "sizeof(CUndergraduateStudent)=" <<
         sizeof(CUndergraduateStudent) << endl;
      return 0;
}
```



输出结果：

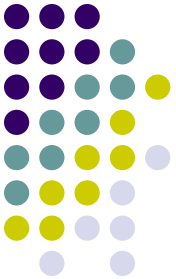
```
Harry Potter qulified for baoyan
Name:Harry Potter
ID:118829212
Age:19
Gender:M
Department:Computer Science
sizeof(string)=4
sizeof(CStudent)=16
sizeof(CUndergraduateStudent)=20
```



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。
  - 类C，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



- 如果写了一个 CMan 类代表男人，后来又发现需要一个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。

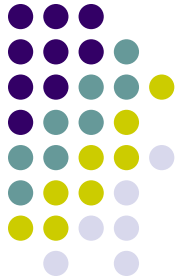
- 好的做法是概括男人和女人共同特点，写一个CHuman类，代表“人”,然后CMan和CWoman都从CHuman派生。

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

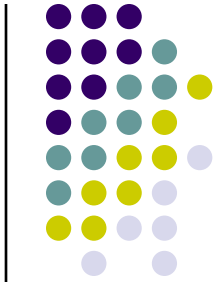
```
class CPoint
{
    double x,y;
    friend class CCircle; //便于Ccirle类操作其圆心
};
class CCircle
{
      double r;
       CPoint center;
};
```

复合关系的使用



- 如果要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）

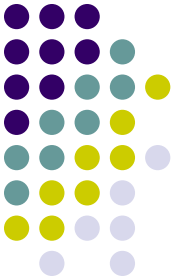
复合关系的使用



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

对不对？

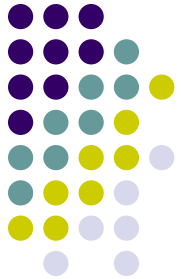
复合关系的使用



- 不好的写法：
  为“狗”类设一个“业主”类的成员对象；
  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;
};
```

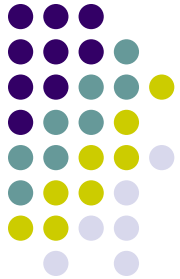
复合关系的使用



- 好的写法：
  为“狗”类设一个“业主”类的对象指针；
  为“业主”类设一个“狗”类的对象数组。

```
class CMaster;    //CMaster必须提前声明，不能先
                  //写CMaster类后写Cdog类
class CDog
{
    CMaster * pm;
};
class CMaster
{
    CDog dogs[10];
};
```

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：

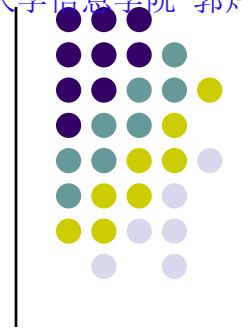
```
class base {
        int j;
public:
        int i;
        void func();
};
```

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

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

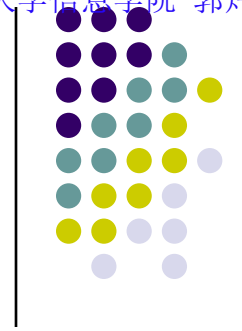
Obj占用的存储空间

➢一般来说，基类和派生类不定义同名成员变量。



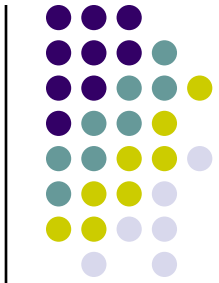
另一种存取权限说明符：protected

- 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
- 基类的protected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
  - 派生类的成员函数可以访问当前对象的基类的保护成员



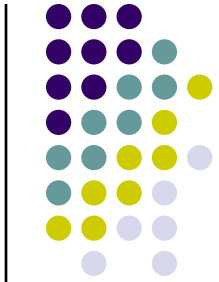
保护成员

```
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
            Son f;
            f.nProtected = 1; //wrong ， f不是当前对象
      }
};
```



```
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;
}
```

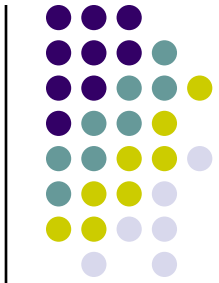
派生类的构造函数



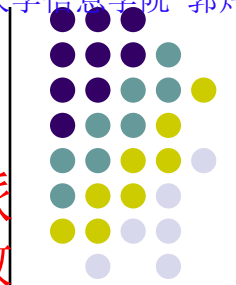
```
class Bug  {
        private :
                int nLegs;     int nColor;
        public:
                int nType;
                Bug (  int legs, int color);
                void PrintBug (){ };
};
class FlyBug: public Bug // FlyBug是Bug的派生类
{
                int nWings;
        public:
     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

表达式中可以出现：FlyBug构造函数的参数、常量



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

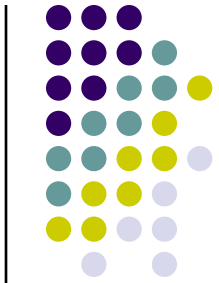


FlyBug fb (2,3,4);

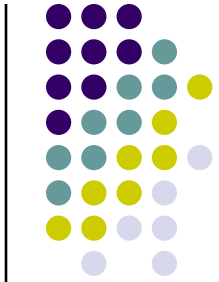
- 在创建派生类的对象时，需要调用基类的构造函数：初始化派生类对象中从基类继承的成员。在执行一个派生类的构造函数之前，总是先执行基类的构造函数。
- 调用基类构造函数的两种方式
  - 显式方式：在派生类的构造函数中，为基类的构造函数提供参数.

    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;   }
```

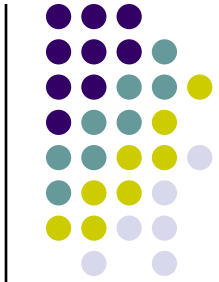


输出结果:



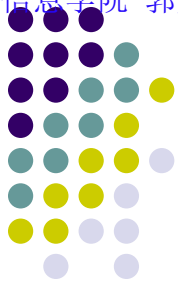
Base 3 constructed
Derived constructed
Derived destructed
Base 3 destructed

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;
}
```

表达式中可以出现：FlyBug构造函数的参数、常量



- 在创建派生类的对象时，在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。
- 派生类的析构函数被执行时，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;

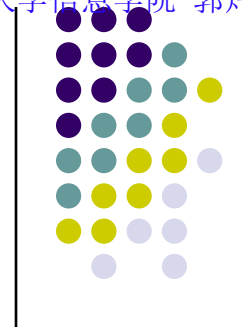
- 如果派生方式是 private或protected，则上述三条不可行。

protected继承和private继承

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

- protected继承时，基类的public成员和protected成员成为派生类的protected成员。
- private继承时，基类的public成员成为派生类的private成员，基类的protected成员成为派生类的不可访问成员。
- protected和private继承不是“是”的关系。

基类与派生类的指针强制转换



- 派生类对象的指针可以直接赋值给基类指针

```
Base * ptrBase = &objDerived;
```

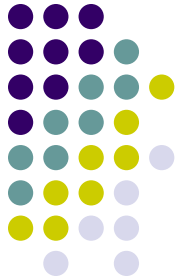
  - ptrBase指向的是一个Derived类的对象；
  - *ptrBase可以看作一个Base类的对象，访问它的public成员直接通过ptrBase即可，但不能通过ptrBase访问objDerived对象中属于Derived类而不属于Base类的成员

- 通过强制指针类型转换，可以把ptrBase转换成Derived类的指针

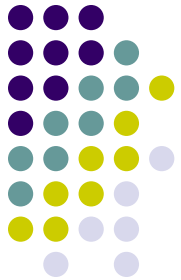
```
Base * ptrBase = &objDerived;
Derived *ptrDerived = (Derived * ) ptrBase;
```

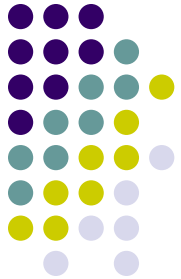
  - 程序员要保证ptrBase指向的是一个Derived类的对象，否则很容易会出错。

```
#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;}
};
```



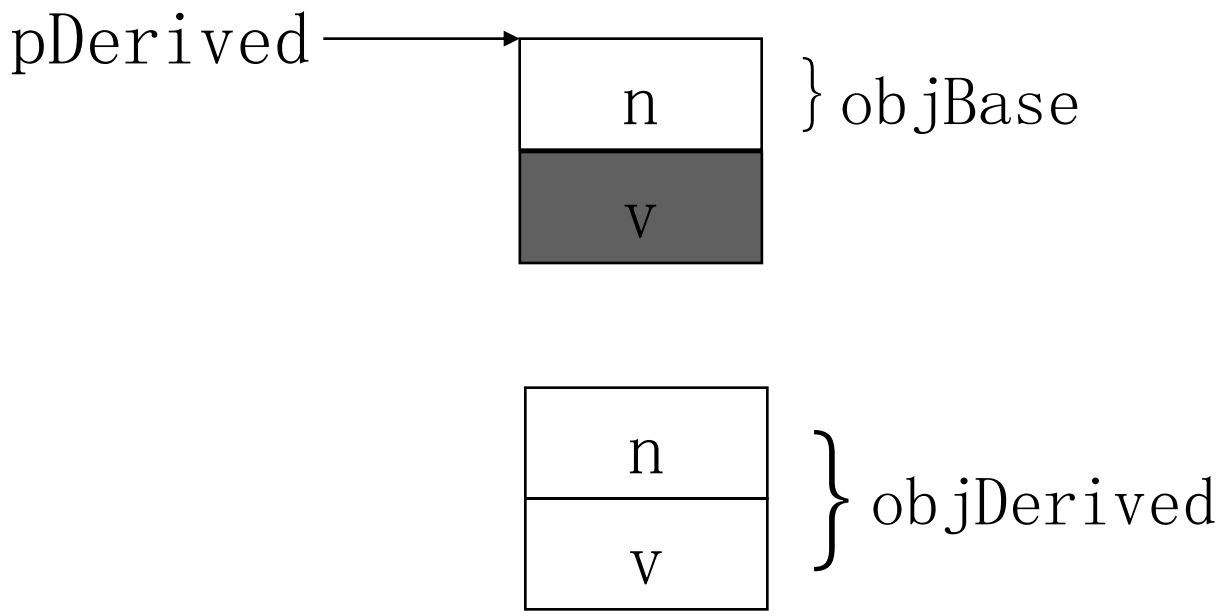
```
class Derived:public Base  {
        public:
                int v;
                Derived(int i):Base(i),v(2 * i) {
                   cout << "Derived constructed" << endl;
                }
                ~Derived()   {
                        cout << "Derived destructed" << endl;
         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





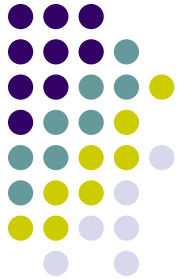
```
int main()  {
        Base objBase(5);
        Derived objDerived(3);
        Base * pBase = & objDerived ;
        //pBase->Func(); //err;Base类没有Func()成员函数
        //pBase->v = 5; //err; Base类没有v成员变量
        pBase->Print();
        //Derived * pDerived = & objBase; //error
        Derived * pDerived = (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，会有问题
        objDerived.Print();
        return 0;
}
```

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

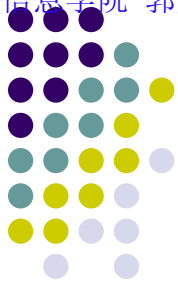


输出结果:

Base 5 constructed
Base 3 constructed
Derived constructed
Base:n=3
Derived:v=1245104 **//pDerived->n 位于别人的空间里**
Derived:n=5
Derived:v=6
Derived:n=3
Derived destructed
Base 3 destructed
Base 5 destructed

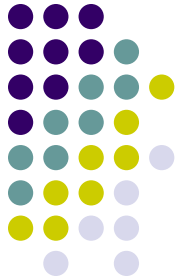


直接基类与间接基类

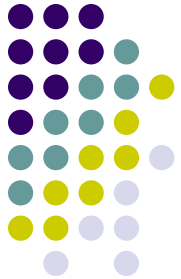


- 类A派生类B，类B可再派生类C，类C派生类D，……
  - 类A是类B的直接基类
  - 类B是类C的直接基类，类A是类C的间接基类
  - 类C是类D的直接基类，类A、B是类D的间接基类
- 在声明派生类时，派生类的首部只需要列出它的直接基类
  - 派生类的首部不要列出它的间接基类
  - 派生类沿着类的层次自动向上继承它的间接基类
  - 派生类的成员包括
    - 派生类自己定义的成员
    - 直接基类中定义的成员
    - 间接基类的全部成员

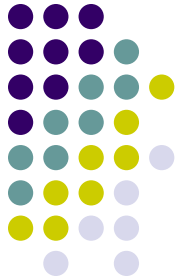
```
#include <iostream>
using namespace std;
class Base {
        public:
                int n;
                Base(int i):n(i)  {
                cout << "Base " << n << " constructed" << endl;
                }
                ~Base() {
                cout << "Base " << n << " destructed" << endl;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

};
```



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

};
```

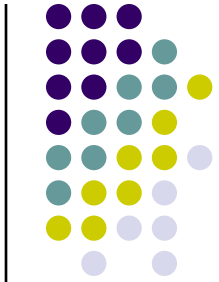


```
class MoreDerived:public Derived {
public:
        MoreDerived():Derived(4) {
                cout << "More Derived constructed" << endl;
        }
        ~MoreDerived() {
                cout << "More Derived destructed" << endl;
        }

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

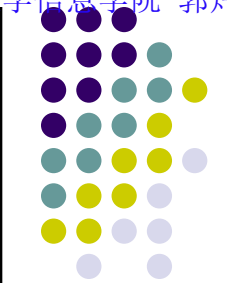


输出结果：

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



多继承



- 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”。

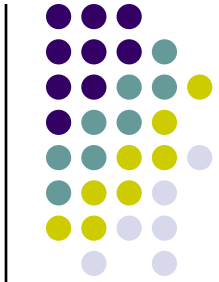
class derived:access-specifier$_1$ base$_1$, access-specifier$_2$ base$_2$, …{

……

};

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

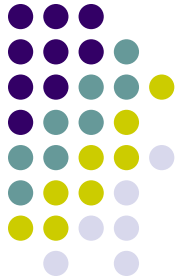
可能需要多继承的例子



- 公司人事管理程序
  - ✓ 销售人员类：
    有销售额属性
  - ✓ 经理类：
    有下属人数属性
  - ✓ 开发人员类：
    。 。 。
  - ✓ 销售经理： 既有销售员的属性，又有经理的属性

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：
  1. 按继承顺序调用基类的构造函数；
  2. 依次调用成员对象的构造函数；
  3. 调用派生类的构造函数。

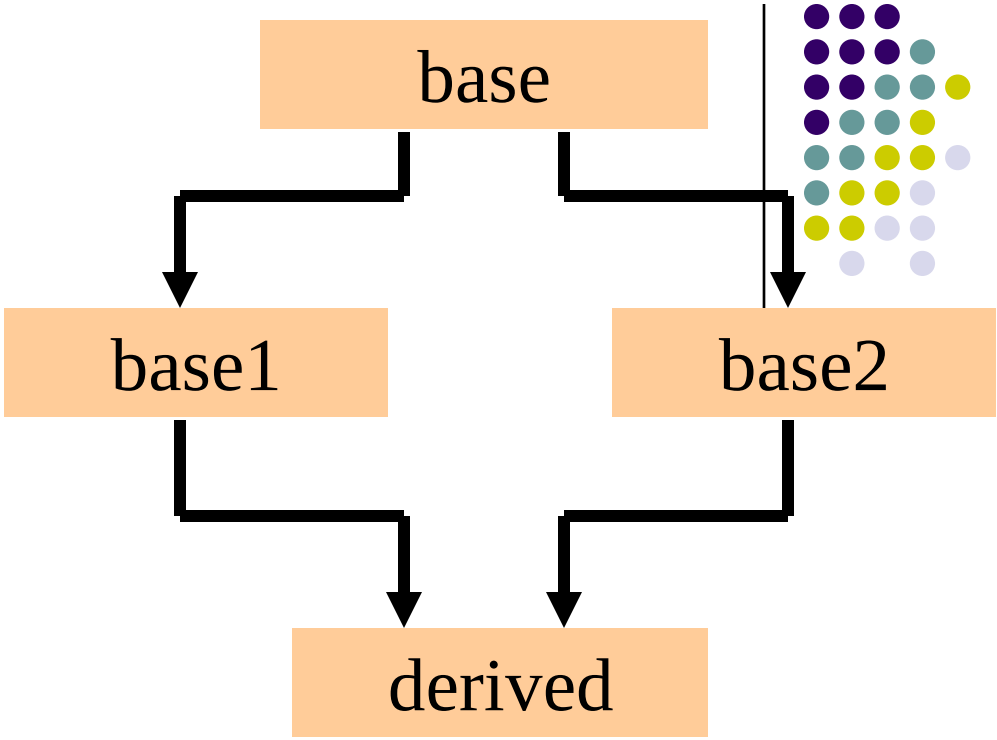
多继承中基类的构造函数调用：

```
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对象的服务

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

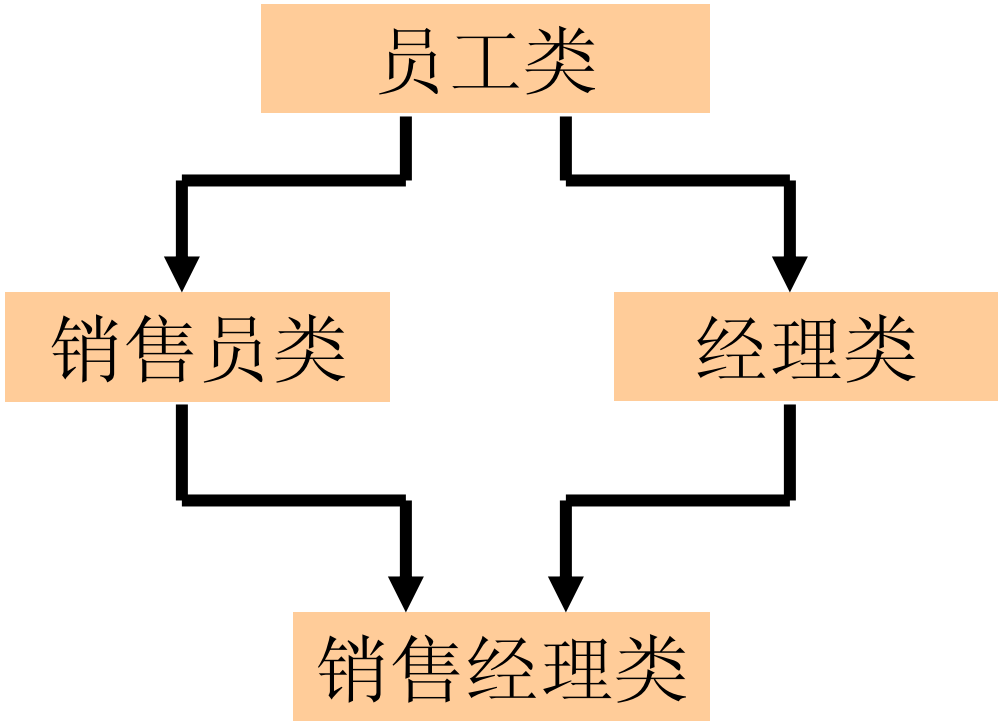
```
int main ()  {
        derived d;
        d.set(10);     //二义 性
        d.base1::set(10);  // error, can't access private member
        d.base2::set(5);

        d.base1::i =5;
        d.base2::i = 5;
}
```

✓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
```

- 因为多继承容易产生二义性等问题，所以使用多继承要慎重。语法上的二义性容易解决，语义上的二义性才是真的麻烦

- 公司人事管理程序



经理类和销售人员类都有“姓名”属性，结果导致销售经理对象中会有两个“姓名”，到底起作用的是哪一个？用“虚继承”解决，略。