# 第五讲 继承

三、把子类对象当作父类对象使用

四、子类中的构造函数、析构函数

# 三、把子类对象当作父类对象使用

把子类对象作为父类对象使用，分三种情况：

（1）将子类对象赋值给父类对象

把等号右边的子类对象，赋值给等号左边的父类对象时，仅仅是把子类中来自于父类的那一部分成员，赋值给等号左边的父类对象，而赋值完成后，等号左边的对象，依然是父类对象。

```
Drived d;
Base b;
b = d;

Base* p = new Base;
*p = d;
delete p;
```

（2）父类引用指向子类对象

- 子类对象用于初始化父类引用

```
Derived d;
Base & b = d;
Derived::func(){}
d.func();
b.func(); //不能使用父类引用调用子类扩展成员
```

- 将子类对象传递给参数为父类引用的函数

```
Base& fun( Base& b ){
   Derived* p= new Derived;
   return *p;
}

Derived d;
Base& b = fun(d);
delete &b;
```

- 函数原型返回值类型为父类引用，而实际return的是子类对象

```
Base& fun(Base& b )
{
    Derived* p= new Derived;
    return *p;
}

Derived d;
Base& b = fun(d);
delete &b;
```

父类引用指向子类对象时，当通过父类引用去调用被子类覆盖的函数时，父类版本的函数将被调用。

(3) 父类指针指像子类对象

- 子类对象的地址分配给父类指针

```
Derived d;
Base* b = &d;
Drived::func(){}
d.func();
b->func();//可以吗?

Derived  * d2 = &d;
Base* b2 = d2;
```

- 将子类对象(地址)传递给参数为父类指针的函数

```
void fun(Base* b)
{
   Derived* p= new Derived;
}

Derived d;
Base* b = fun(&d);
```

- 函数原型返回值类型为父类指针，而实际return的是子类指针

```
Base* fun( Base* b )
{
   Derived* p= new Derived;
   return p;
}

Derived d;
Base* b = fun(&d);
delete b;
```

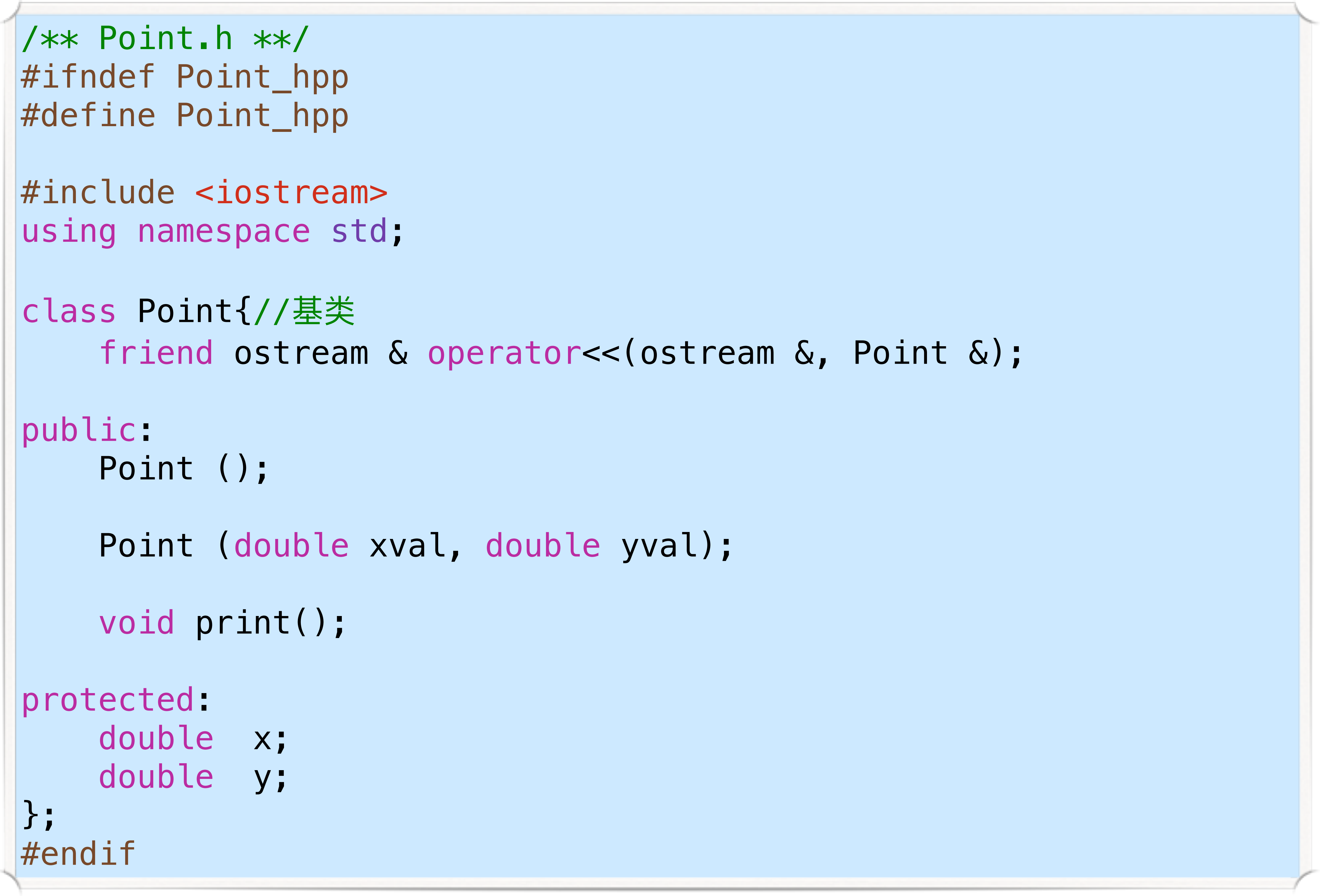
可以把子类对象当作父类对象使用，但不能把父类对象当作子类对象使用。

```
Base b;
Derived d;
d = b;           // Error
Derived* p = &b; // Error
Derived& r = b;  // Error
```

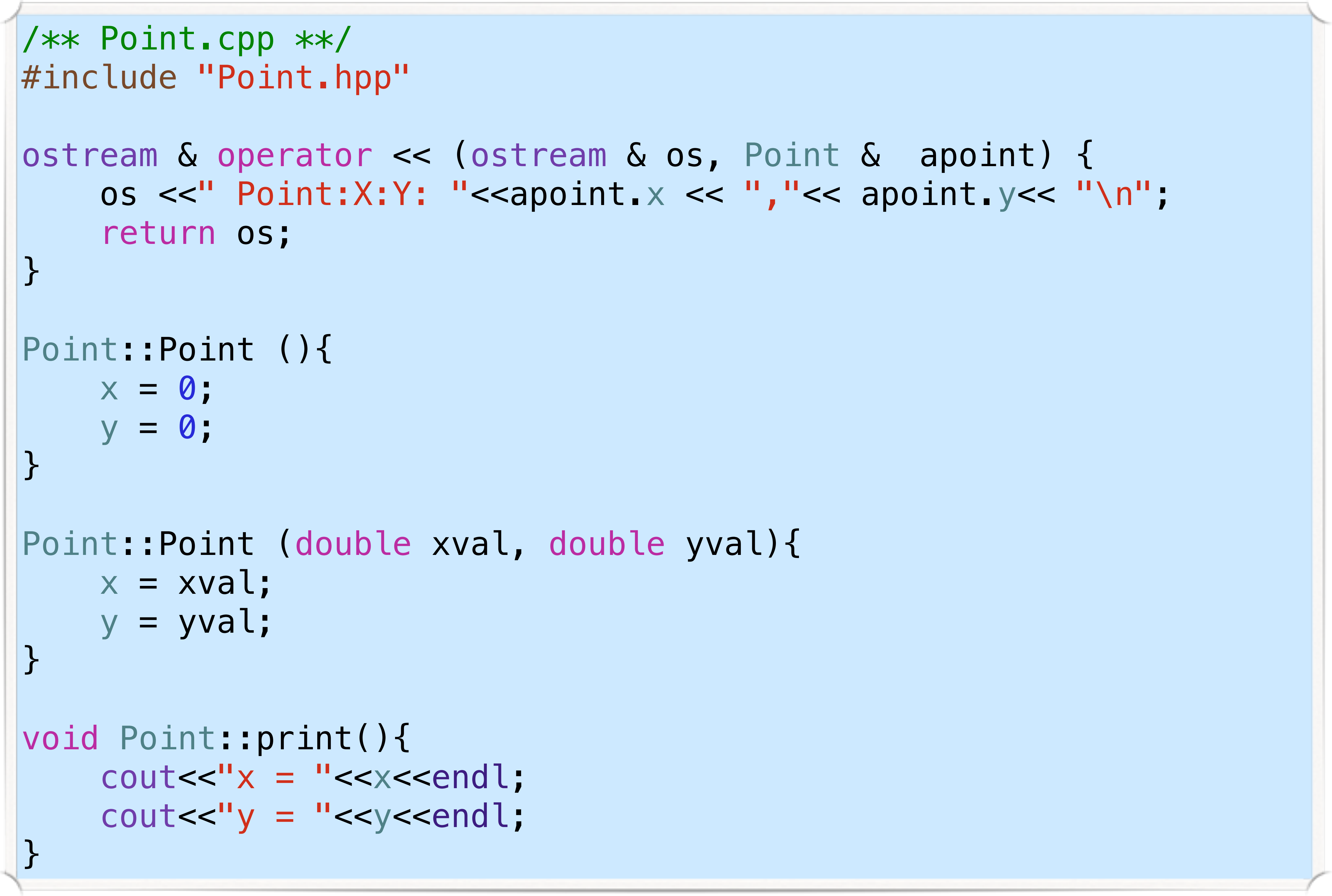
如果父类指针指向子类对象，需要把父类指针转换成指向的子类对象，来访问子类扩展成员。这就是所谓的向下转型。但是，向下转型是危险的，因为父类指针有可能并没有指向子类对象。

```
Base * b = new Derived;
Derived::func(){}

//b->func();//error
((Derived* )b)->func(); //ok
```

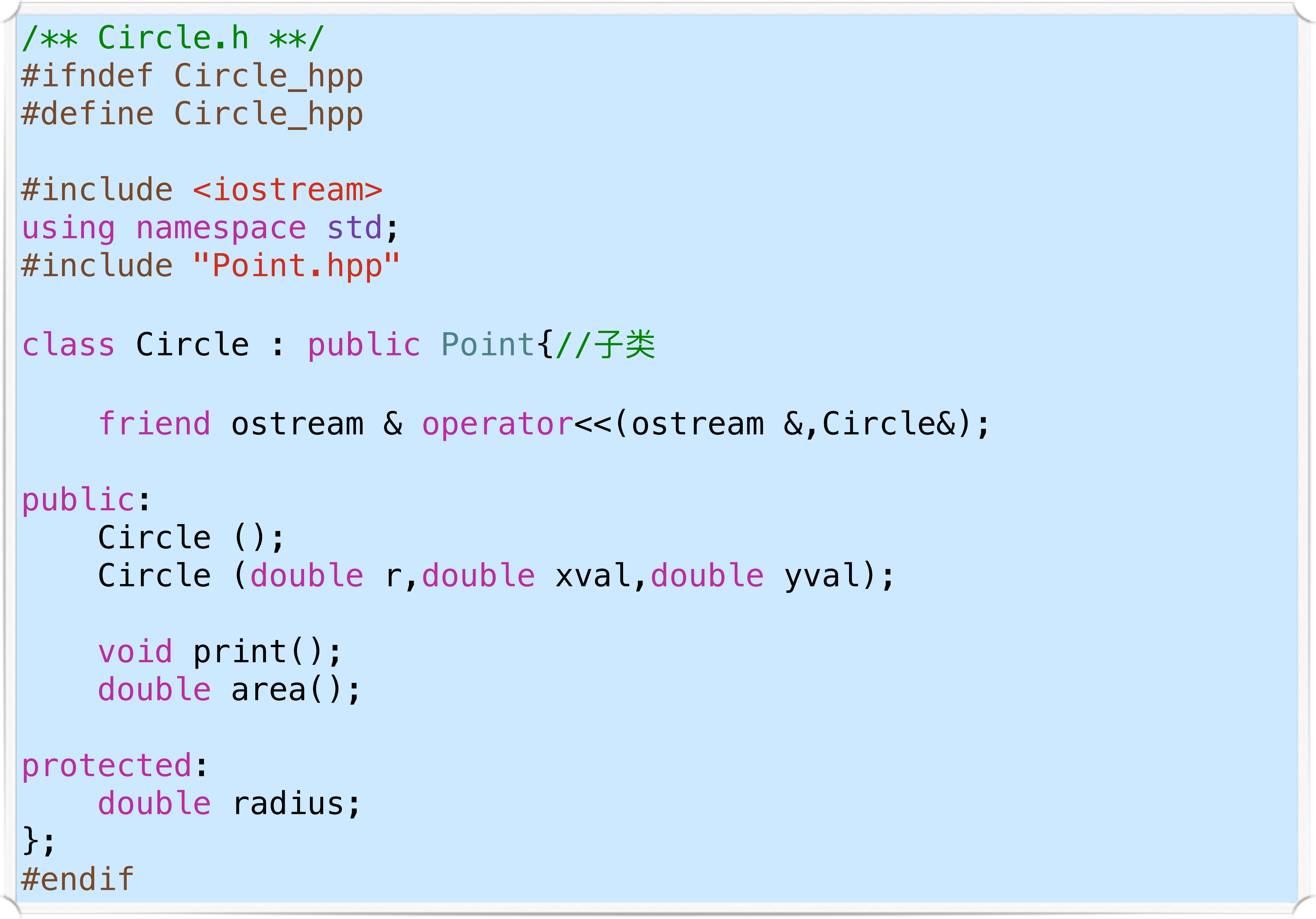
【例3-1】子类对象当父类对象使用示例



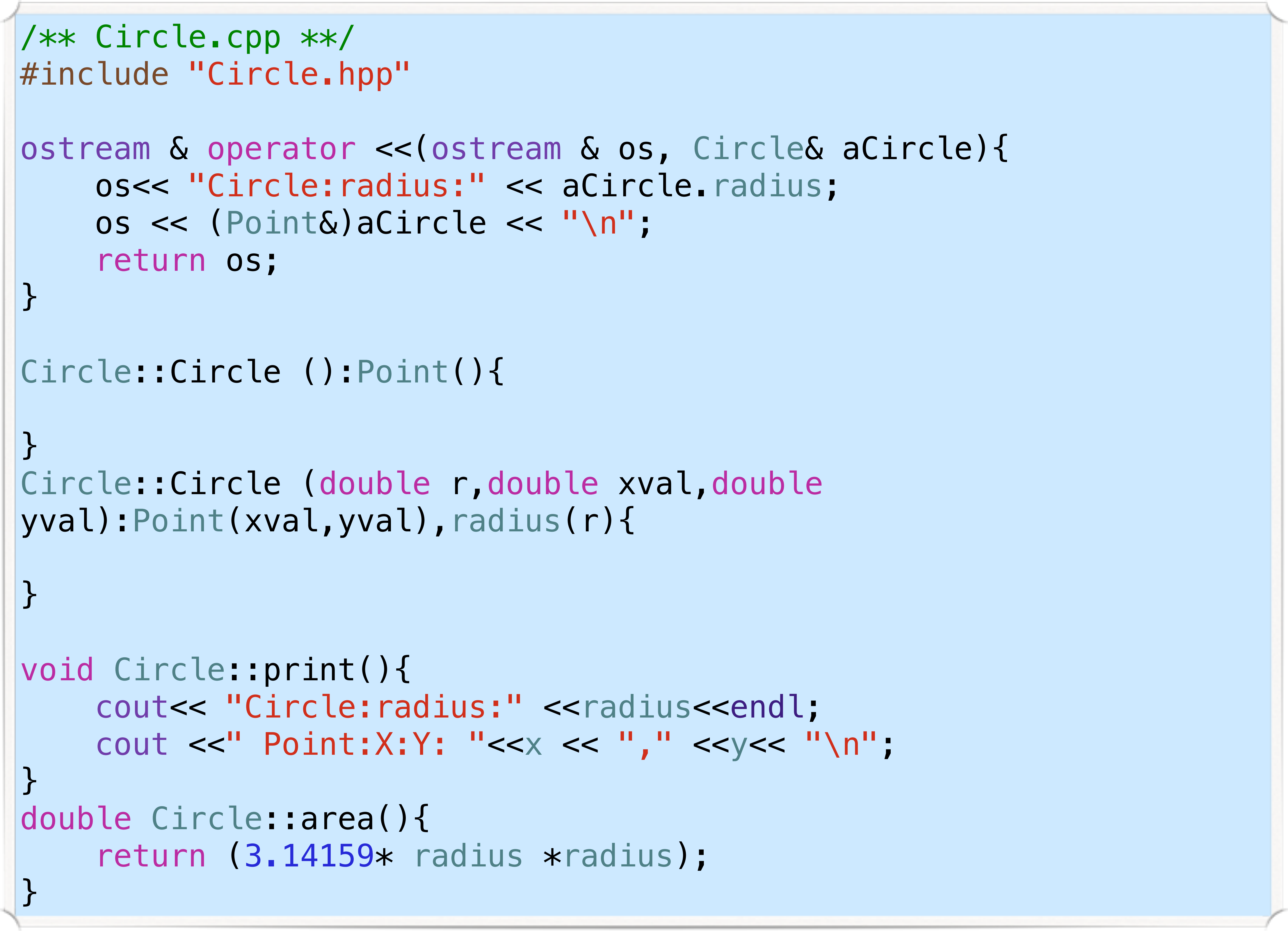
```
/** Point.h **/
#ifndef Point_hpp
#define Point_hpp

#include <iostream>
using namespace std;

class Point{//基类
    friend ostream & operator<<(ostream &, Point &);

public:
    Point ();

    Point (double xval, double yval);

    void print();

protected:
    double  x;
    double  y;
};
#endif
```



```
/** Point.cpp **/
#include "Point.hpp"

ostream & operator << (ostream & os, Point &  apoint) {
    os <<" Point:X:Y: "<<apoint.x << ","<< apoint.y<< "\n";
    return os;
}

Point::Point (){
    x = 0;
    y = 0;
}

Point::Point (double xval, double yval){
    x = xval;
    y = yval;
}

void Point::print(){
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;
}
```



```cpp
/** Circle.h **/
#ifndef Circle_hpp
#define Circle_hpp

#include <iostream>
using namespace std;
#include "Point.hpp"

class Circle : public Point{//子类

    friend ostream & operator<<(ostream &,Circle&);

public:
    Circle ();
    Circle (double r,double xval,double yval);

    void print();
    double area();

protected:
    double radius;
};
#endif
```



```
/** Circle.cpp **/
#include "Circle.hpp"

ostream & operator <<(ostream & os, Circle& aCircle){
    os<< "Circle:radius:" << aCircle.radius;
    os << (Point&)aCircle << "\n";
    return os;
}

Circle::Circle ():Point(){

}
Circle::Circle (double r,double xval,double
yval):Point(xval,yval),radius(r){

}

void Circle::print(){
    cout<< "Circle:radius:" <<radius<<endl;
    cout <<" Point:X:Y: "<<x << "," <<y<< "\n";
}
double Circle::area(){
    return (3.14159* radius *radius);
}
```

```
/** main.cpp  Part-1 **/
#include "Point.hpp"
#include "Circle.hpp"

int main(int argc, const char * argv[]) {

    Point p(2,3);
    cout <<"Point P=  "<< p;

    Point pp(0,0);
    cout <<"Point PP=  "<< pp;

    Circle c(7,6,5);
    cout <<"Circle c=  "<< c;

    pp = p;//同类型对象的赋值
    cout <<"Point PP=  "<< pp;

    pp = c; //把子类对象赋值给父类对象
    cout <<"Point PP=  "<< pp;
}
```

程序运行结果如下:

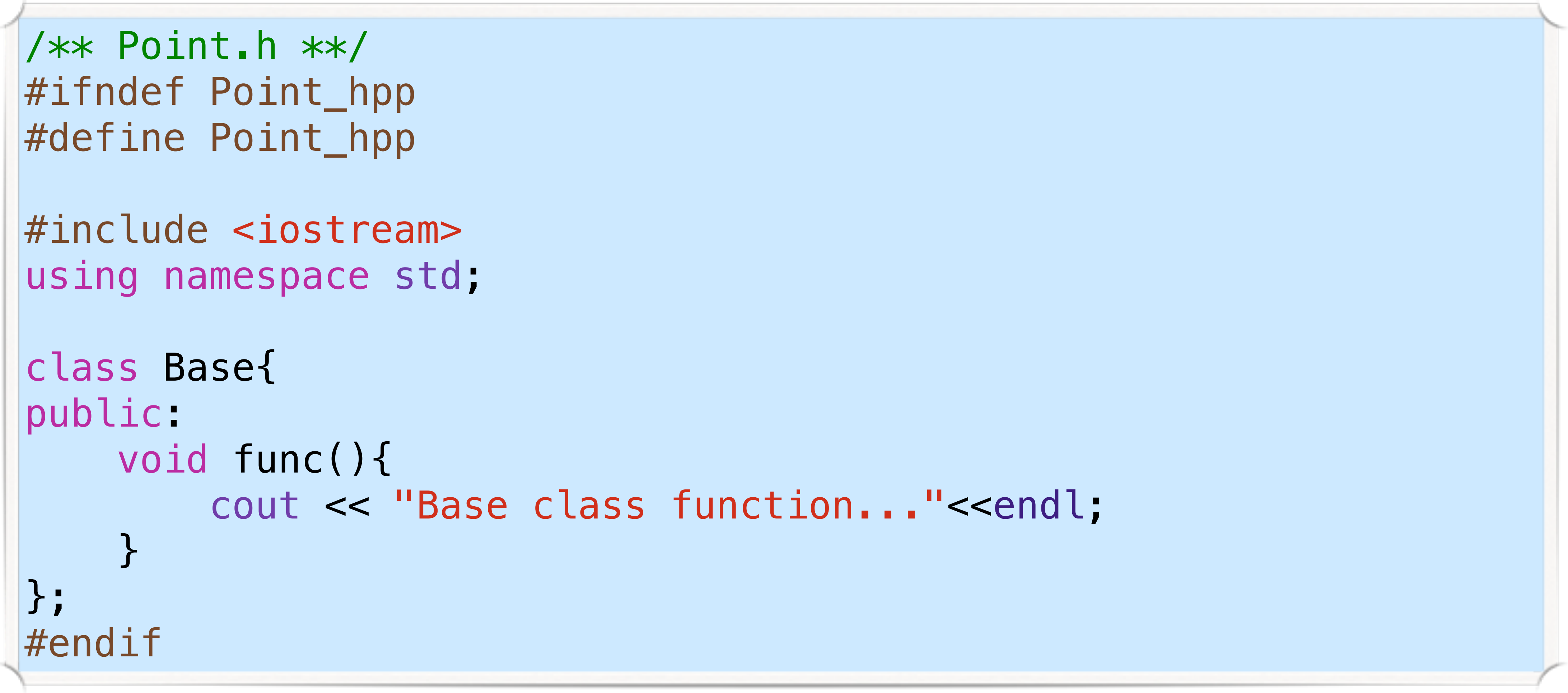
```
    Point P=   Point:X:Y: 2,3
    Point PP=   Point:X:Y: 0,0
    Circle c=  Circle:radius:7
    Point:X:Y: 6,5

    Point PP=   Point:X:Y: 2,3
    Point PP=   Point:X:Y: 6,5
```

```cpp
/** main.cpp  Part-2 **/
int main(int argc, const char * argv[]) {

    pp= (Point)c;//先转型，再赋值。向上转型是安全的。
    cout <<"Point PP=  "<< pp;

    //c = (Circle) pp;不能把父类对象，赋值给子类
    //c=pp;

    Point*  pPoint;
    pPoint = &c;//父类指针指向子类对象

    pPoint ->print();//调用父类版本
    ((Circle*)pPoint)->print();//向下转型

    // Circle* pc = &pp;

    Point& r = c; //父类引用，指向子类对象。
    r.print();    //调用父类版本print

    ((Circle&)r).print();
}
```

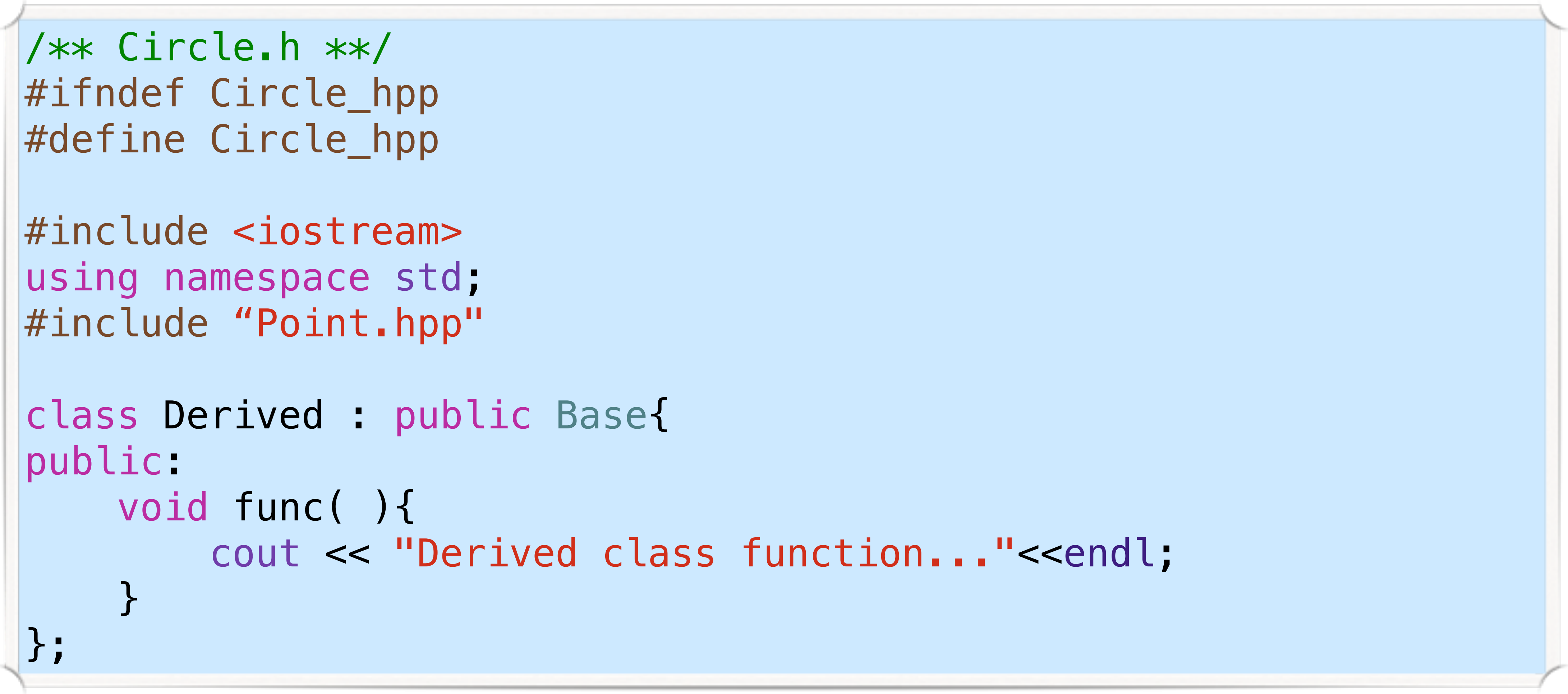
程序运行结果如下：

```
    Point PP= Point:X:Y: 6,5
    x = 6
    y = 5
    Circle:radius:7
    Point:X:Y: 6,5
    x = 6
    y = 5
    Circle:radius:7
    Point:X:Y: 6,5
```

【例3-2】父类版本子类版本方法调用示例



```
/** Point.h **/
#ifndef Point_hpp
#define Point_hpp

#include <iostream>
using namespace std;

class Base{
public:
    void func(){
        cout << "Base class function..."<<endl;
    }
};
#endif
```



```
/** Circle.h **/
#ifndef Circle_hpp
#define Circle_hpp

#include <iostream>
using namespace std;
#include “Point.hpp"

class Derived : public Base{
public:
    void func( ){
        cout << "Derived class function..."<<endl;
    }
};
```

```cpp
/** main.cpp **/

void foo(Base & b){
    b.func();
}

int main(int argc, const char * argv[]){

    Derived d;
    Base b;
    Base* p = &d;
    Base& br = d;

    b = d;
    b.func();
    d.func();
    p->func();
    foo(d);
    br.func();

    return 0;
}
```

程序运行结果如下：

```
    Base class function...
    Derived class function...
    Base class function...
    Base class function...
    Base class function...
```

# 4.1 子类中的构造、析构和赋值运算符重载函数

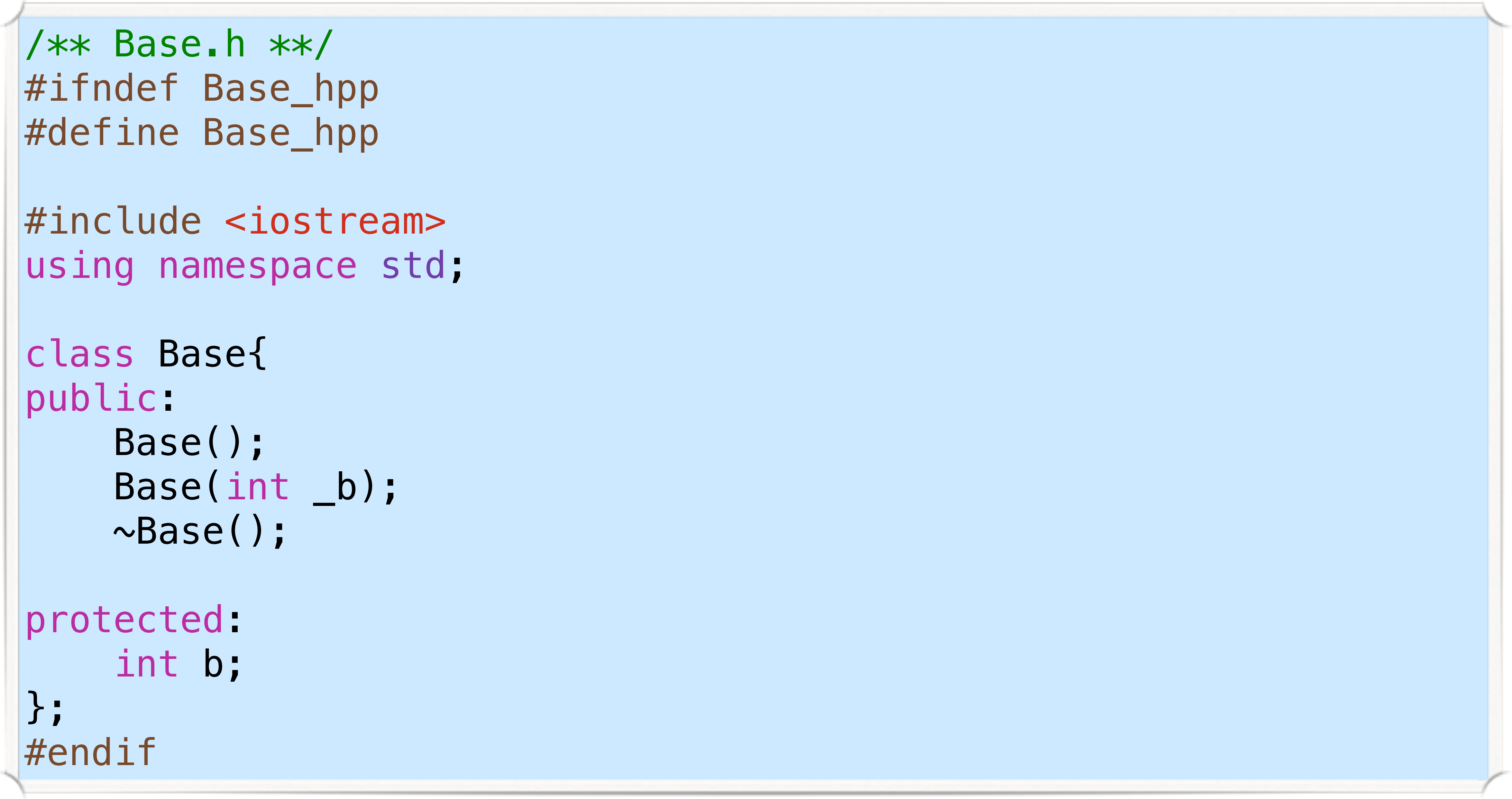
由于构造函数、析构函数不能被继承，如果需要的话，可以在子类中定义。

在子类中构造一个对象时，先调用父类的构造函数来构造子类继承自父类部分。派生类的构造函数只需要初始化自己扩展部分。

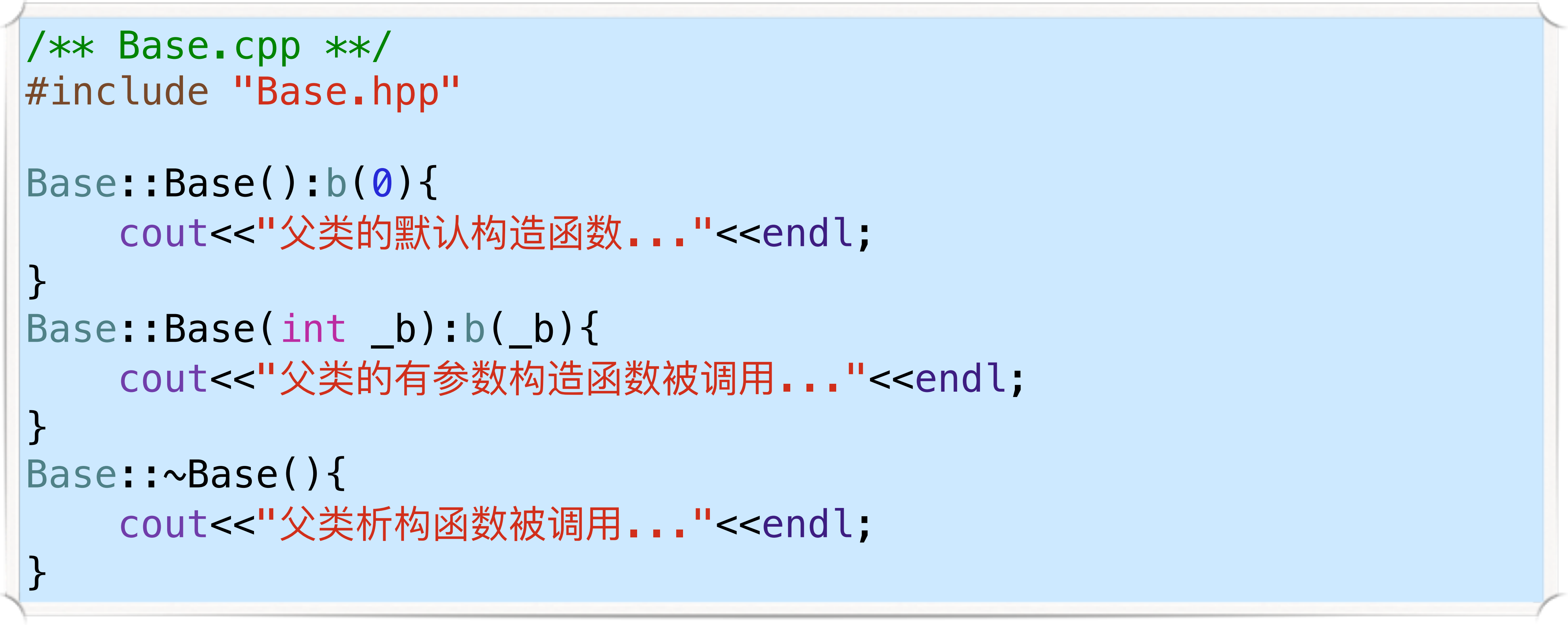
如果基类的构造函数需要任何参数，它们必须在派生类构造函数的参数列表中提供。

由于析构函数不需要任何参数，调用子类的析构函数会自动调用父类的析构函数以完成其工作后。

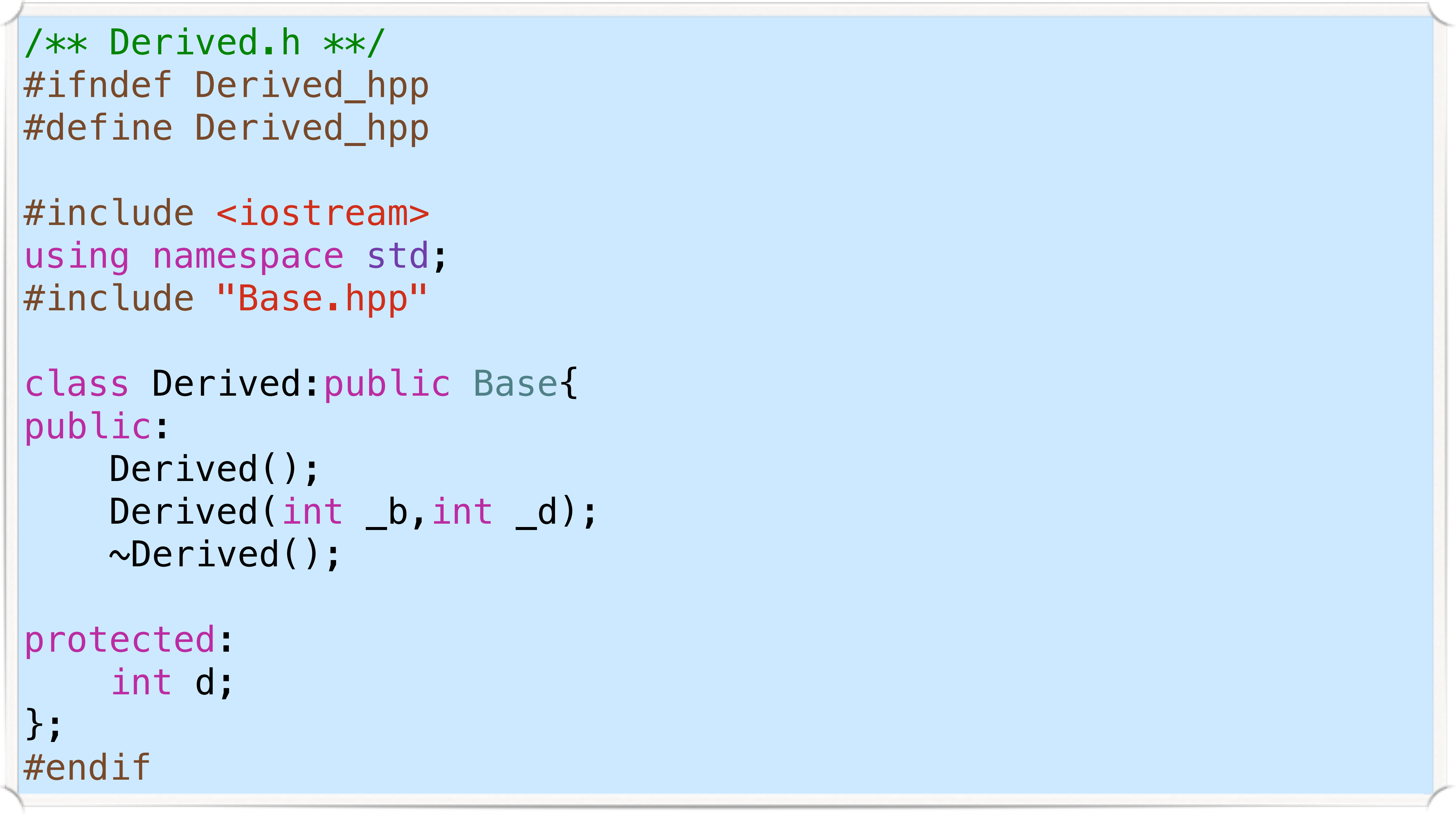
## 【例4-1】子类中的构造函数、析构函数



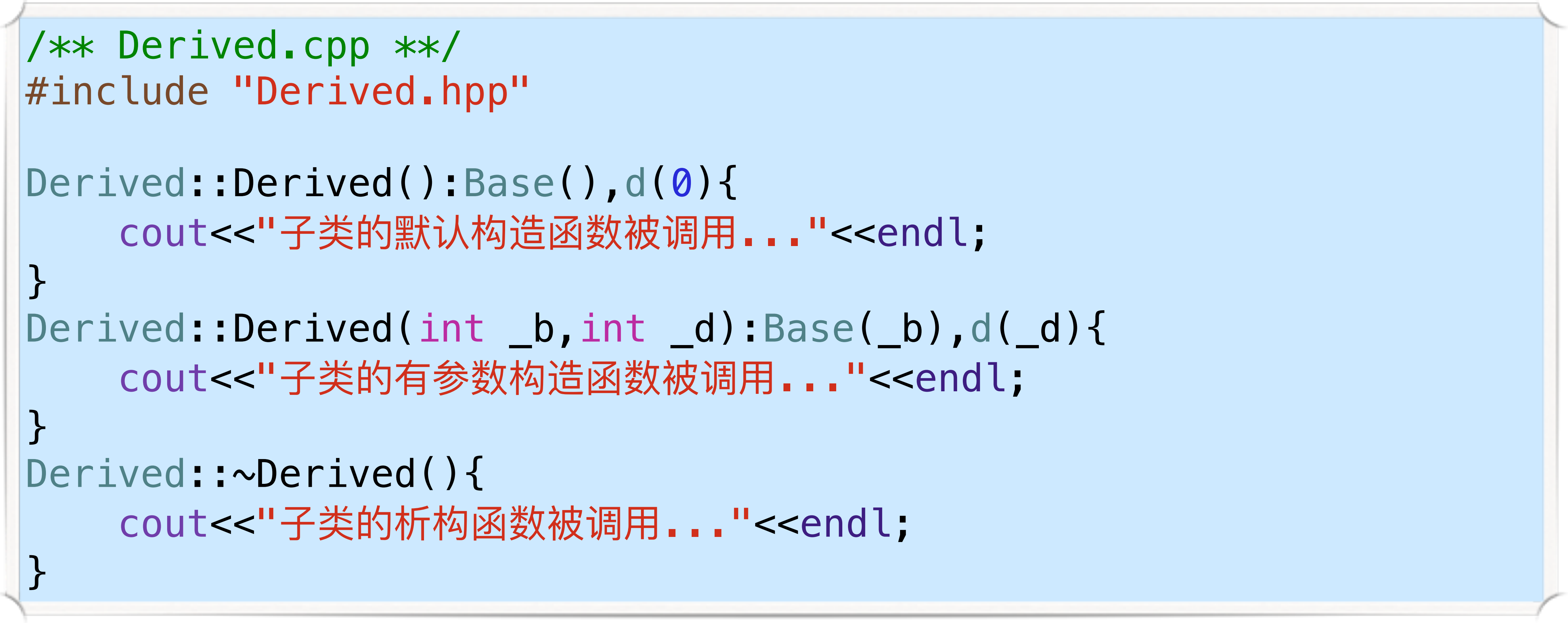
```
/** Base.h **/
#ifndef Base_hpp
#define Base_hpp

#include <iostream>
using namespace std;

class Base{
public:
    Base();
    Base(int _b);
    ~Base();

protected:
    int b;
};
#endif
```



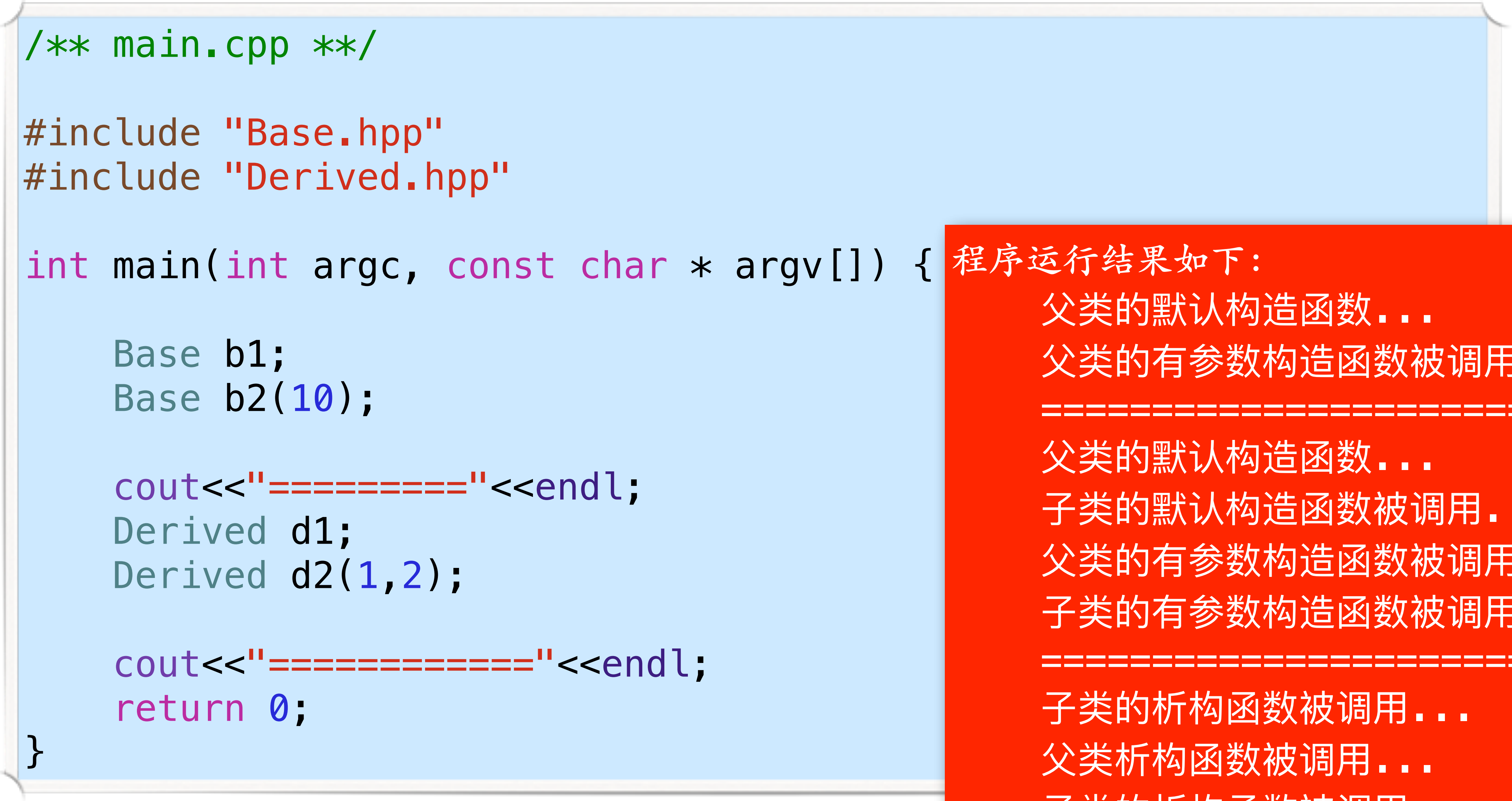
```cpp
/** Base.cpp **/
#include "Base.hpp"

Base::Base():b(0){
    cout<<"父类的默认构造函数..."<<endl;
}
Base::Base(int _b):b(_b){
    cout<<"父类的有参数构造函数被调用..."<<endl;
}
Base::~Base(){
    cout<<"父类析构函数被调用..."<<endl;
}
```



```
/** Derived.h **/
#ifndef Derived_hpp
#define Derived_hpp

#include <iostream>
using namespace std;
#include "Base.hpp"

class Derived:public Base{
public:
    Derived();
    Derived(int _b,int _d);
    ~Derived();

protected:
    int d;
};
#endif
```



```
/** Derived.cpp **/
#include "Derived.hpp"

Derived::Derived():Base(),d(0){
    cout<<"子类的默认构造函数被调用..."<<endl;
}
Derived::Derived(int _b,int _d):Base(_b),d(_d){
    cout<<"子类的有参数构造函数被调用..."<<endl;
}
Derived::~Derived(){
    cout<<"子类的析构函数被调用..."<<endl;
}
```



```
/** main.cpp **/

#include "Base.hpp"
#include "Derived.hpp"

int main(int argc, const char * argv[]) {

    Base b1;
    Base b2(10);

    cout<<"========="<<endl;
    Derived d1;
    Derived d2(1,2);

    cout<<"============"<<endl;
    return 0;
}
```

```
程序运行结果如下：
    父类的默认构造函数...
    父类的有参数构造函数被调用...
    ==============================
    父类的默认构造函数...
    子类的默认构造函数被调用...
    父类的有参数构造函数被调用...
    子类的有参数构造函数被调用...
    ==============================
    子类的析构函数被调用...
    父类析构函数被调用...
    子类的析构函数被调用...
    父类析构函数被调用...
    父类析构函数被调用...
    父类析构函数被调用...
```

构造子类时，先执行父类的构造函数，然后执行子类的构造函数。

析构子类时，先执行子类的析构函数，然后执行父类的析构函数。