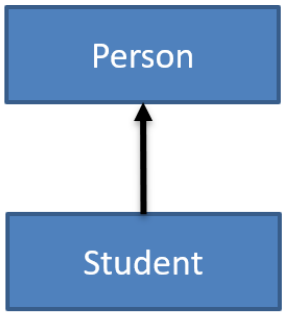
Upcasting




```
Student s;
Person p = s;
Person* ptr = &s;
Person &pr = s;
```

准确的说就是通过将一个派生类里的对象以它的基类对象去应用，通常这个基类对象是一个指针类型的对象

具体的写法如下

```cpp
#include <iostream>
using namespace std;

class Base {
public:
    virtual void display() {
        cout << "Display Base" << endl;
    }
};

class Derived : public Base {
public:
    void display() {
        cout << "Display Derived" << endl;
    }
};

int main() {
    Derived derivedObj;
    Base* basePtr = &derivedObj; // Upcasting

    basePtr->display(); // Calls Derived's display, thanks to virtual function

    return 0;
}
```

在main函数中创建出来一个派生类对象`Derived derivedObj`，

然后创建出一个基类对象`Base* basePtr = &derivedObj; // Upcasting`

这个步骤就是upcasting，注意这里是用到的引用，注意写法

baseptr实际上是一个指针，是派生类对象的引用，有自身类的属性

现在就是涉及到函数重写的部分了

在基类中的函数前加上virtual，使得自身的函数可以重写，

之后在派生类中的同名函数就实现了这个函数的重写

`basePtr->display();`

用上述的语句访问到重写后的函数

当然这个`basePtr->display();`也可以用函数的形式表达

当存在多个派生类对象的时候

用以下的代码

```
class Base {
public:
    void display() {
    cout << "base" << endl;
    }
};

class Derived1 : public Base {
public:
    void display() {
    cout << "derived1" << endl;
    }
};

class Derived2 : public Base {
public:
    void display() {
    cout << "derived2" << endl;
    }
};
void func(Base& x) {
    x.display();
}

int main() {
    Base x;
    Derived1 y;
    Derived2 z;
    func(x);
    func(y);
    func(z);
    return 0;
}
```

注意那个`func(Base& x)`的写法，里面x传入了之后其实就已经是upcasting的过程

前面的Base&就相当于是`Base* basePtr = &derivedObj;`

可以看上面的图

两种写法都可以

那么现在用了upcasting之后就会存在一些问题

就是当销毁那个Base* basePtr对象的时候，到底调用的是哪一个析构函数呢

首先在程序执行结束之前肯定不会全部销毁

那么当执行`delete basePtr`的时候，实际上调用的是基类的析构函数

因为它本身其实算是基类的对象

那么这个时候派生类的数据不会被销毁，这个时候在程序执行完毕之前可能会造成数据泄露

这个时候我们就要加入一个虚析构函数来防止这种现象的发生

具体操作如下

```
#include <iostream>
using namespace std;

class Base {
public:
    // 声明虚拟析构函数
    virtual ~Base() {
        cout << "Base destructor called" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor called" << endl;
    }
};

int main() {
    Base *basePtr = new Derived();
    delete basePtr;  // 首先调用 Derived 的析构函数，然后是 Base 的析构函数
    return 0;
}
```

1. **对于普通继承，析构函数执行的顺序是什么？**
    - 在普通继承中，析构函数的执行顺序是从派生类到基类的。当一个对象被销毁时，首先调用派生类的析构函数，然后是其基类的析构函数。这确保了派生类分配的资源首先被释放，然后是基类分配的资源。
2. **对于子类型多态（向上转型），析构函数执行的顺序是怎样的？**
    - 在子类型多态或向上转型的情况下，如果我们通过基类的指针删除派生类的对象，那么为了确保派生类的析构函数被调用，基类的析构函数必须是虚拟的。如果基类的析构函数不是虚拟的，那么只有基类的析构函数会被执行，这可能会导致派生类分配的资源未被正确释放，从而引发资源泄露。
3. **如何确保调用的是派生类的析构函数？**

- 为了确保在多态使用中正确调用派生类的析构函数，基类的析构函数必须声明为虚拟的（`virtual`）。这样，当通过基类的指针或引用删除对象时，会调用正确的析构函数，即最具体派生类的析构函数。