# 封装性

## 类和对象

C++将对象抽象成类进行处理，由于目的的不同抽象出的信息也可能不同。

根据目的不同，决定哪些信息需要暴露出去，哪些信息需要隐藏起来，这些通过访问限定符（public、protected、private）。如果需要暴露信息使用public访问控制符，如果需要隐藏信息则使用private访问控制符。

## 对象实例化

类就是一个模板，所谓的对象实例化就是根据一个类的设定，制造多个对象的过程。

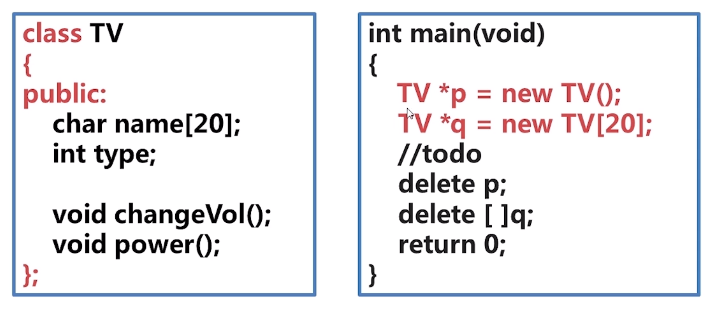
有两种实例化对象的方式：

1. 从栈中实例化：



从栈中实例化对象不需要手动释放内存，系统完成这部分工作，但是在堆中实例化则需要手动释放内存。

1. 从堆中实例化：



在堆中实例化对象需要手动释放内存，如果实例化的是一个对象，则采用delete p的形式，如果是实例化对象数组，那么采用delete []p的形式。

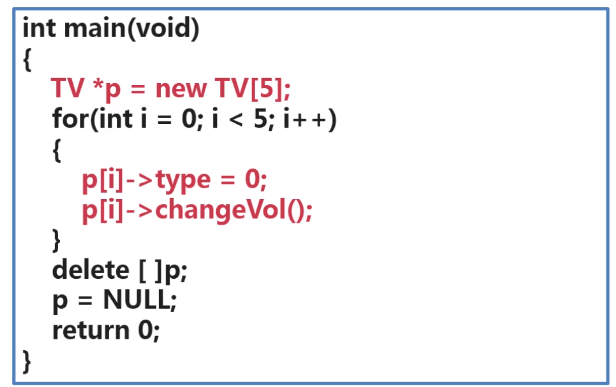
## 对象成员访问

实例化对象后需要访问对象的成员，根据栈和堆两种实例化方式的不同，访问成员的方式也有所不同。

针对单一对象而言访问方式：

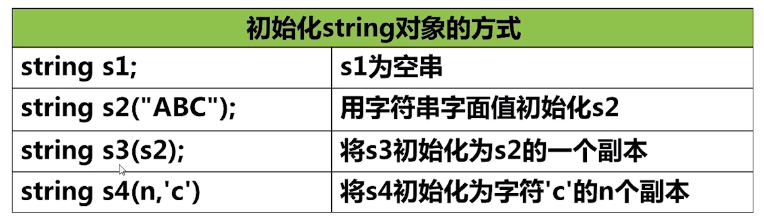


针对对象数组而言访问方式：



## String

### 初始化



### 常用操作



有一种连接方式需要格外注意：



只有string变量与双引号“”的字符串结合的时候才是合法的，两个“”字符串结合是非法的，比如这里的s6。

拓展：为什么Java的String设置为不可变的呢？

Java中将String设计成不可变的是综合考虑到各种因素的结果，想要理解这个问题，需要综合内存，同步，数据结构以及安全等方面的考虑：

1、字符串常量池的需要

字符串常量池(String pool, String intern pool, String保留池) 是Java堆内存中一个特殊的存储区域，当创建一个String对象时，假如此字符串值已经存在于常量池中，则不会创建一个新的对象，而是引用已经存在的对象。

假若字符串对象允许改变，那么将会导致各种逻辑错误，比如改变一个对象会影响到另一个独立对象。严格来说，这种常量池的思想,是一种优化手段。

2、允许String对象缓存HashCode

Java中String对象的哈希码被频繁地使用，比如在hashMap 等容器中。

字符串不变性保证了hash码的唯一性，因此可以放心地进行缓存。这也是一种性能优化手段，意味着不必每次都去计算新的哈希码。

3、安全性

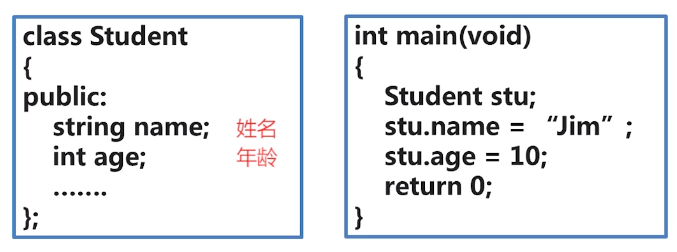
String被许多的Java类(库)用来当做参数。例如：网络连接地址URL，文件路径path，还有反射机制所需要的String参数等，假若String不是固定不变的，将会引起各种安全隐患。

## 数据的封装

所谓数据的封装就是将类中成员变量的操作转换为成员函数的操作，这样才是封装性的体现，即不直接暴露成员变量本身，而是暴露出成员变量操作的函数。

面向对象的思想就是以谁做什么为基础，反映到代码层面上就是类的成员函数。

如下例子就是违背了这种思想：



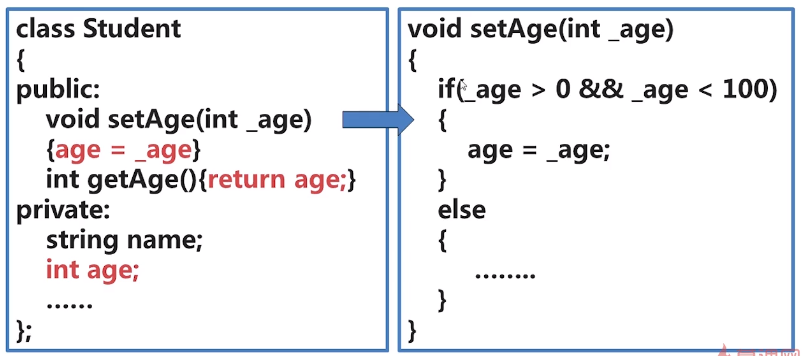
应该采用如下的方式：



这样具备两种好处：

1、更加符合面向对象的思想；

2、可以根据需要对参数进行校验：

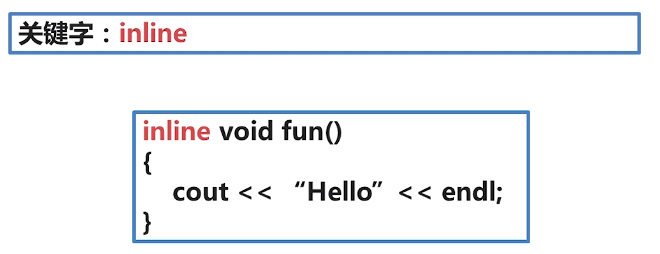


3、可以设置只读属性：

有些时候我们可能需要设置某些成员变量只能被读或者写操作，这样就单独设置仅有getter或者setter中的某一个操作函数即可。



## 内联函数



内联函数与普通函数的区别：

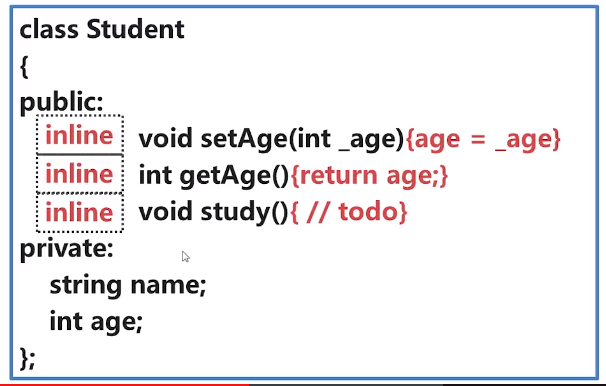


内联函数相当于省去了2、4调用和返回的过程，只有一个3函数本身执行的过程。这样效率更高，速度更快，但是仅仅适用于结构比较简单的函数，如果函数逻辑复杂，即使设置内联函数，编译器也会拒绝按照内联函数的方式处理。

## 类内/类外定义

### 类内定义

如果将成员函数和函数体写在类的内部，这就是类内定义。类内定义的成员函数，编译器会优先将其编译为inline函数，它不会显式地标识inline，而是优先按照inline编译。对于复杂的函数，无法编译为内联函数的，则编译为普通函数。

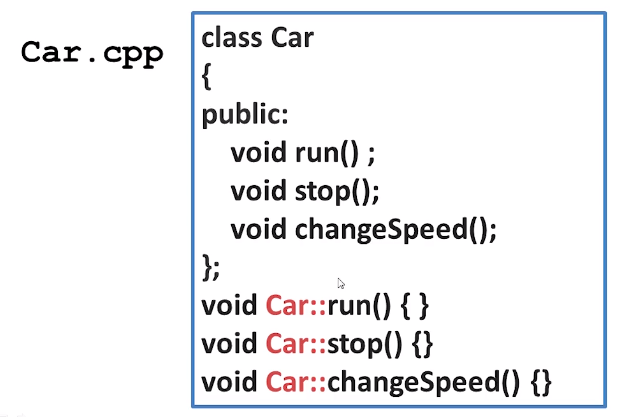


既然将这个函数实现定义在类内部，系统就默认为这个函数比较短小，可以采用inline方式编译，如果是分开的那么就不会默认为该函数非常短小了。

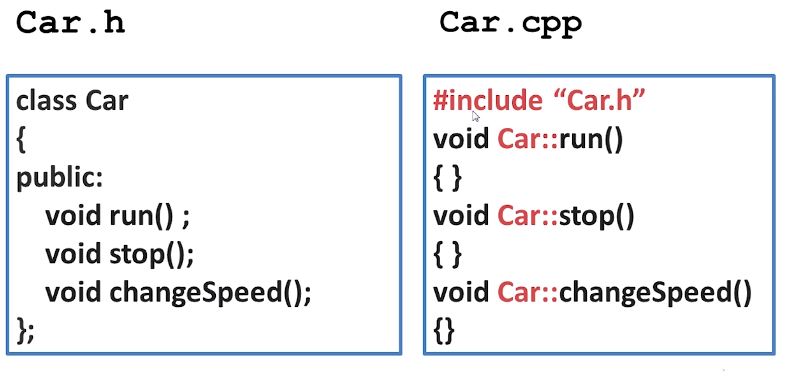
### 类外定义

类外定义是成员函数和函数体写在类的外面。包括两种：同文件类外定义和分文件类外定义。

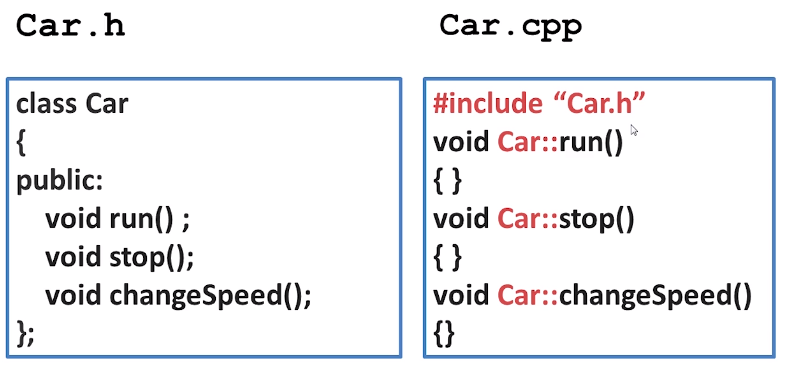
同文件类外定义：



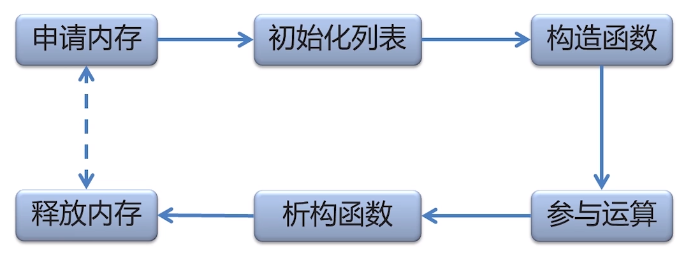
分文件类外定义（常用）：



专业的C++项目都是按照分文件定义的方式来组织代码的。首先，要定义一个头文件.h，这里声明成员变量和成员函数；然后，在.cpp文件中导入.h文件（即声明），实现具体函数。



## 构造函数



### 内存分布

要想知道对象是怎么存储的，需要知道内存按照用途是怎么划分的：



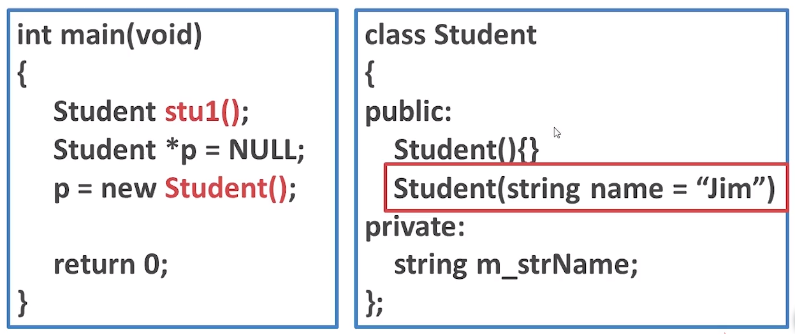
### 对象初始化

一个对象如果new之后没有进行设置，那么就相当于我们知道了地址但是没有内容，所以需要先进行对象的初始化操作。

对象的初始化包括：有且仅有一次，根据条件初始化。

有且仅有一次的初始化就是构造函数，构造函数没有返回值，可以重载，但是实例化对象时只用到其中一个构造函数。

在实例化对象的时候，如果不传入参数，则成为默认的构造函数，如下：



### 初始化列表



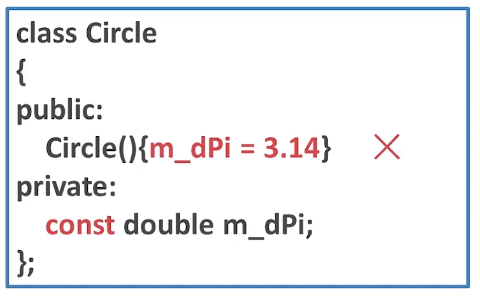
初始化列表的一些基本特性：

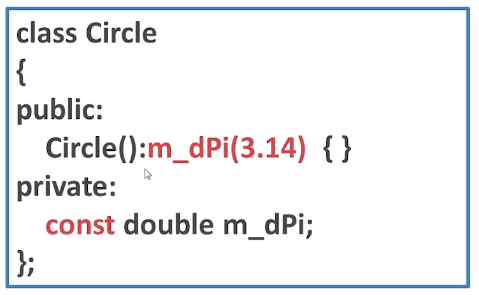
1、初始化列表先于构造函数执行；

2、初始化列表只能用于构造函数；

3、初始化列表可以同时初始化多个数据成员。

既然初始化列表的工作可以由构造函数完成，那么为什么还需要初始化列表呢？



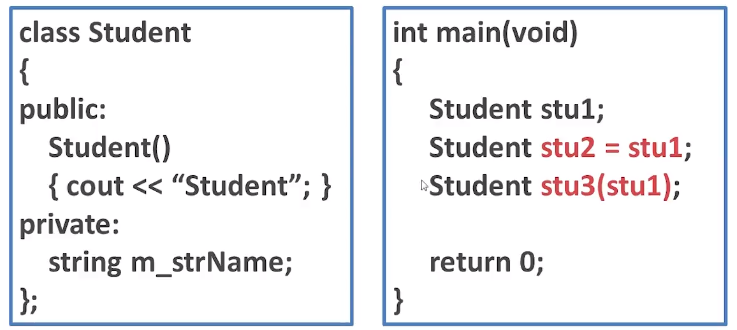


实例：





### 拷贝构造函数



拷贝构造函数定义：

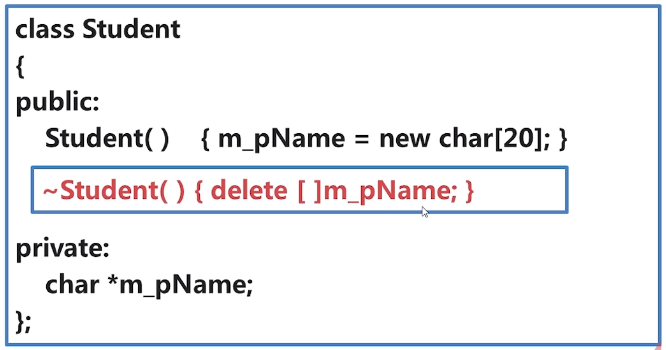


说明：

1、如果没有自定义的拷贝构造函数则系统默认自动生成一个默认的拷贝构造函数；

2、当采用直接初始化或者复制初始化实例化对象时系统自动调用拷贝构造函数。

## 析构函数

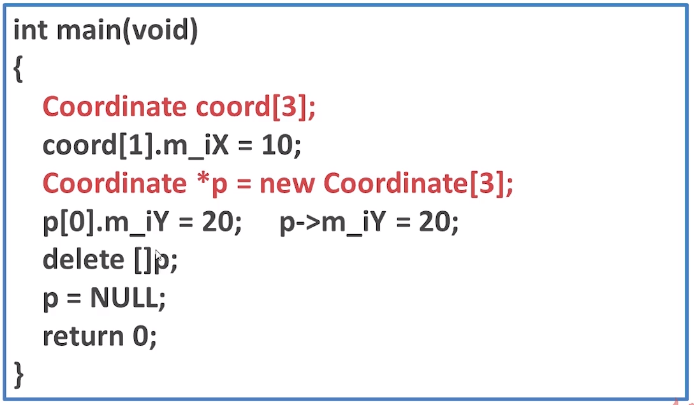


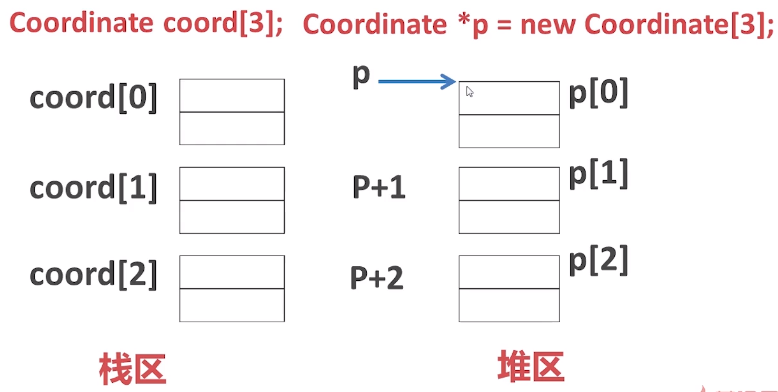
说明：

析构函数没有参数不允许重载！

## ----

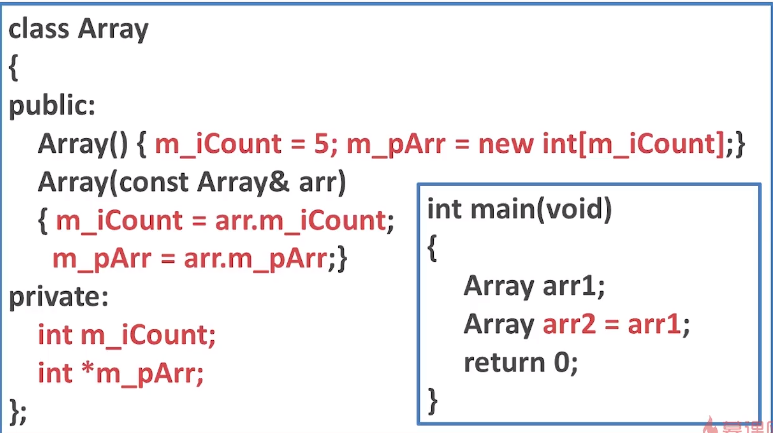
## 对象数组

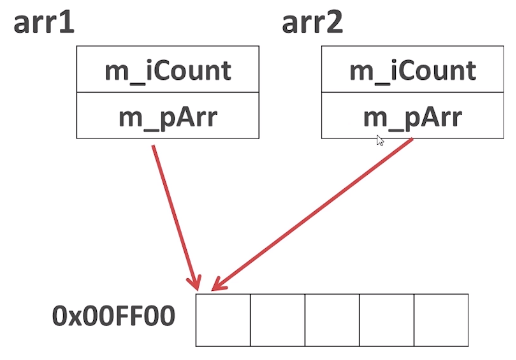




## 深浅拷贝

### 浅拷贝





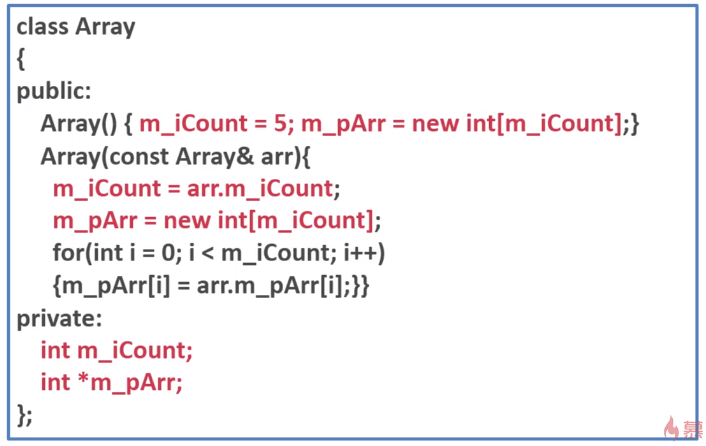
浅拷贝只是将数据成员的值进行简单的拷贝（不管是不是指针），上述arr1和arr2会同时指向一个内存地址，如果往arr1写入一个值，然后再往arr2写入另一个值，那么这个地址对应的值就会被重写覆盖原来的值，这显然不是我们期望的。还有更严重的，在销毁arr1的时候，为了避免内存泄露，会释放m\_pArr的内存，同理arr2在销毁的时候也会释放这个内存，这样同一个内存地址释放了两次，计算机会core掉。

我们期望的是内存地址指向的数据依次执行拷贝，即如下：

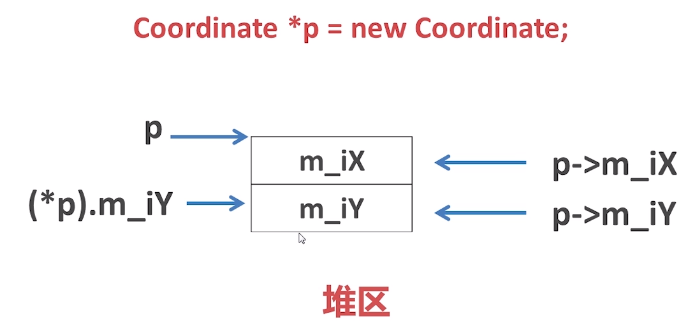


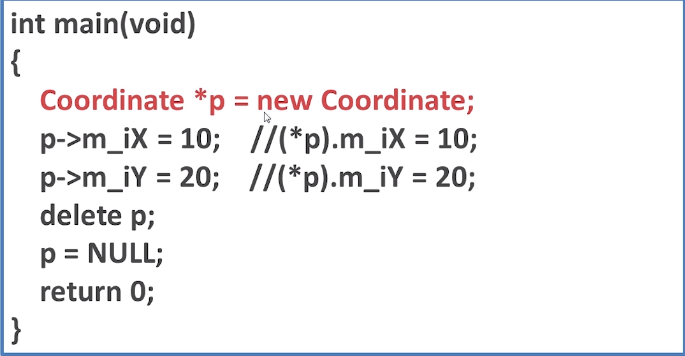
### 深拷贝

深拷贝不是简单的执行拷贝，而是将堆中的内存数据执行拷贝。



## 对象指针

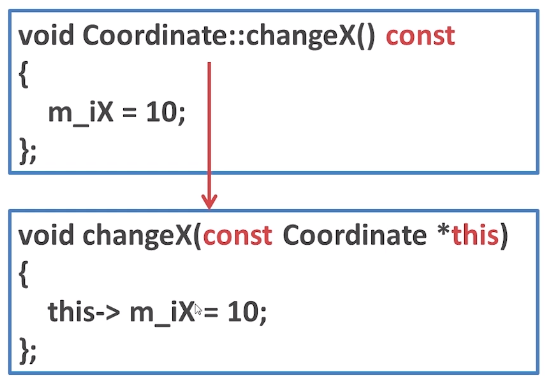




## const

### 常对象成员

### 常成员函数



### 常指针与常引用

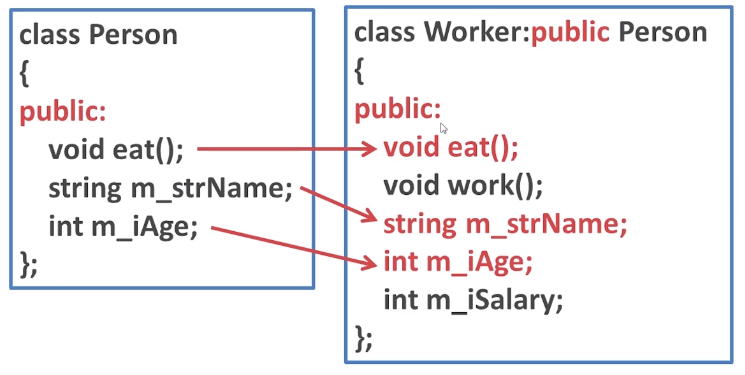
# 继承性

## 公有继承

class A : public B

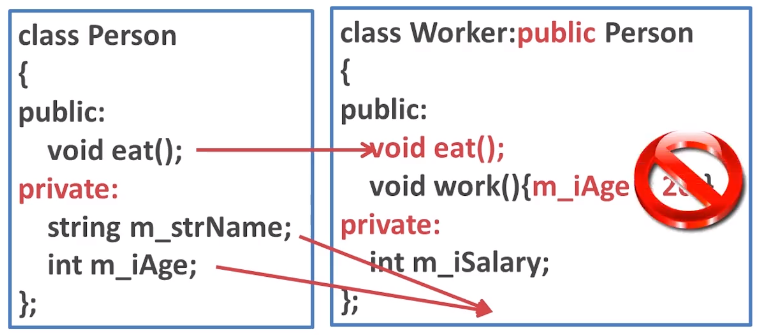


1、public成员变量公有继承

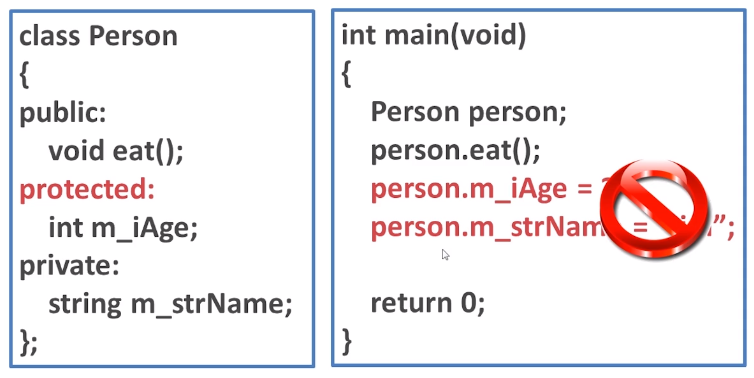


2、private成员变量公有继承

在继承中，private成员变量被继承在子类不可见的位置，而不是private的位置，因此不能在子类中访问父类的private变量。

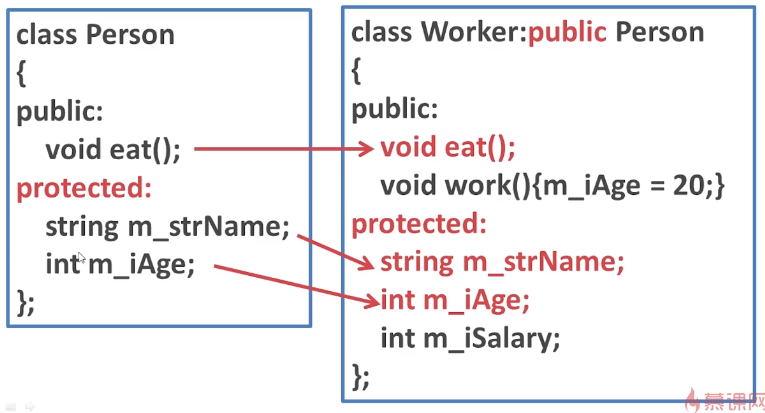


3、protected成员变量继承



说明：当在子类中访问父类protected成员变量时会报错。

在公有继承情况下protected成员变量的访问：



可以通过公有继承的子类的成员函数访问继承来的protected类型的成员变量。

## 私有继承

class A : private B



私有继承是has a的关系，即包含关系，子类对象包含一个父类对象，比如线段和坐标。

## 保护继承

class A : protected B

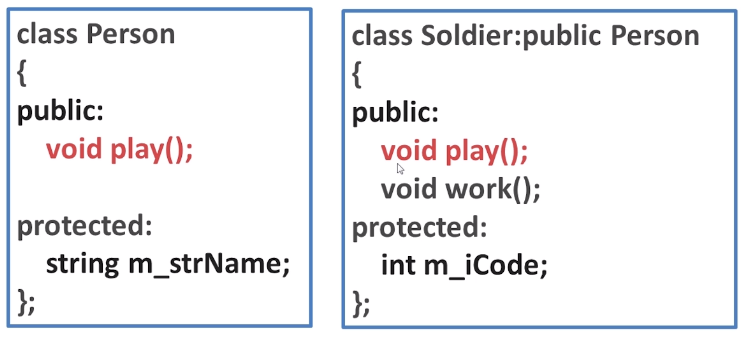


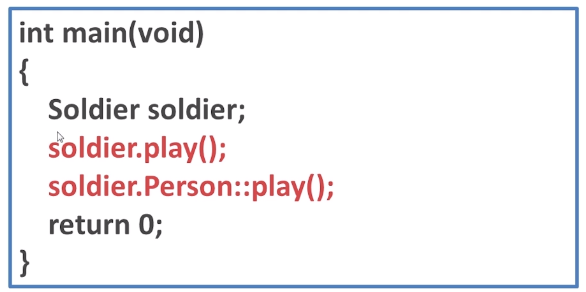
与公有继承不同的是，父类的public成员变量被继承在子类的protected控制符下面。

## 隐藏

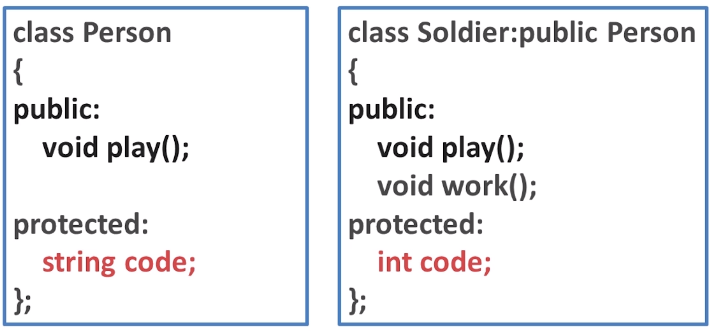
### 同名

如果子类和父类同时定义同一个方法，则其访问方式为：





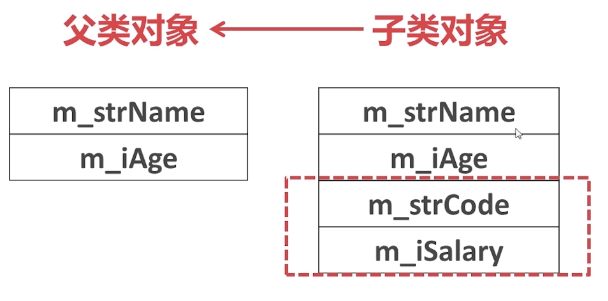
如果子类和父类同时定义了同一个成员变量，则其访问方式为：



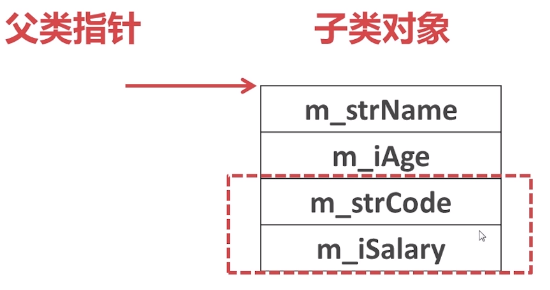


其实针对上述这样的成员变量“重名”的情况，尽量避免使用。也可以采用数据类型不同确定不同名称，比如父类中string类型成员变量命名为m\_scode，子类中int类型成员变量命名为m\_icode。

### is a



子类对象赋值给父类（或者子类初始化父类），其含义就是用子类继承自父类的成员变量给父类中的对应成员变量赋值，父类中不具备子类的成员变量丢失。

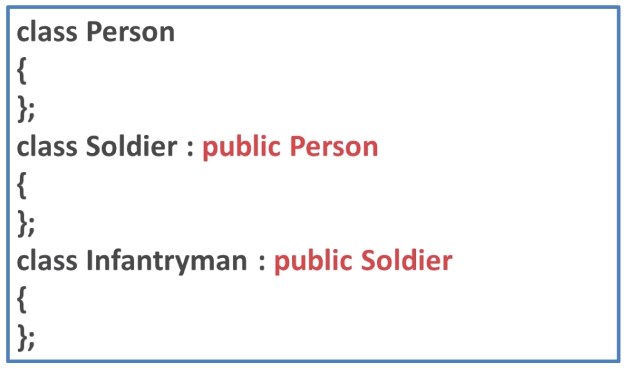


如果父类指针访问子类对象，那么它也只能访问到子类中继承自父类的成员变量，而不能访问子类特有的成员变量。

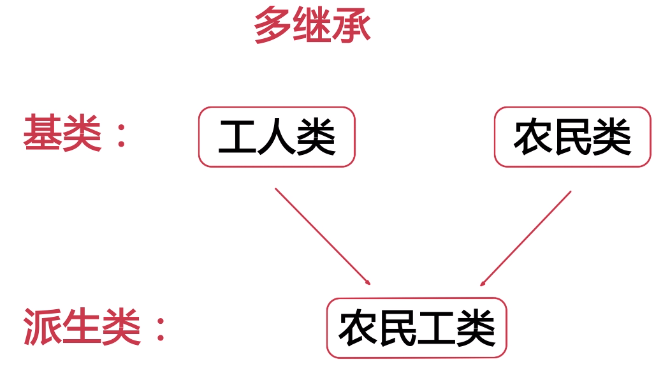
## 多继承/多重继承

多重继承：





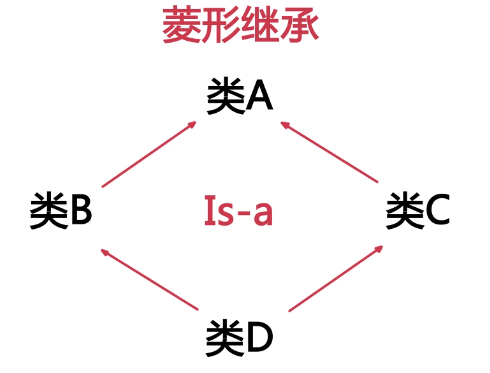
多继承：





## 虚继承

虚继承是为了解决多继承和多重继承带来的问题，比如菱形继承中存在的多个父类的问题：



在类D中存在两个完全一样的类A的副本，这种冗余数据我们不能承担带来的额外的开销，显然不是我们希望的。

虚继承方式：class A : virtual public B

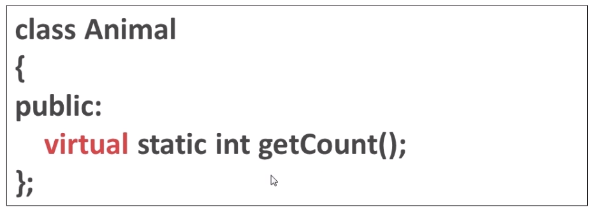
# 多态性

## 虚函数

### 原理

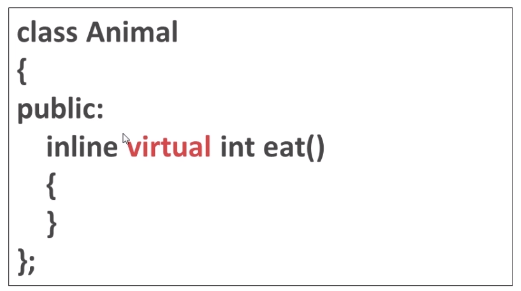
### 限制

1. 不能修饰普通函数（必须是某一类的成员函数，不能是全局函数）；
2. 不能修饰静态成员函数



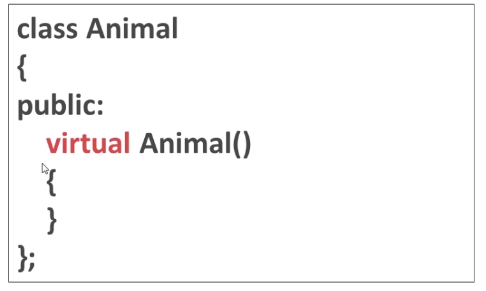
因为static的成员函数不属于任何一个对象，它与类共生存的，所以使用virtual修饰会造成编译报错。

1. 不能修饰内联函数：



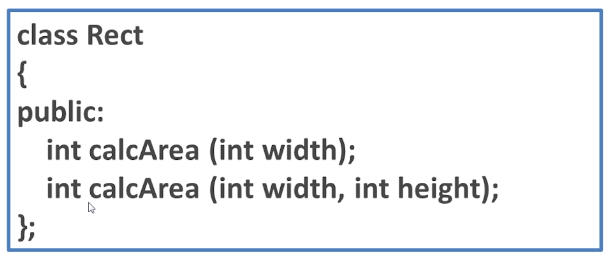
如果修饰内联函数，计算机会忽略inline关键字，而使其变为虚函数。

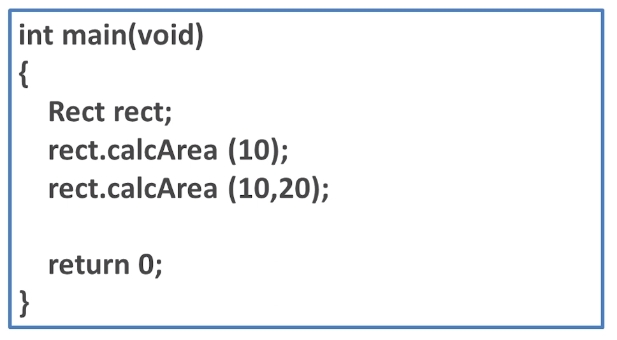
1. 不能修饰构造函数：



### 静态多态

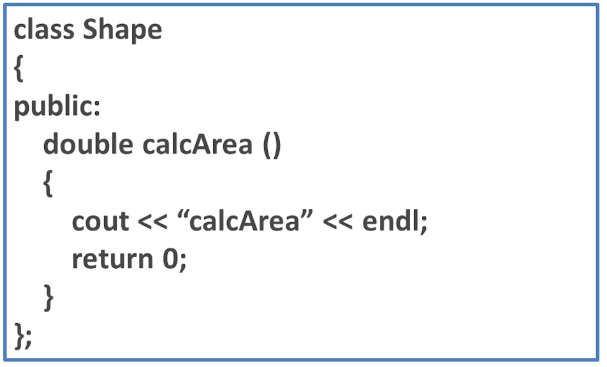
静态多态又叫早绑定，在编译阶段就确定使用哪个函数了，比如重载函数，只要在定义的时候设置了参数就可以知道执行哪一个函数了。

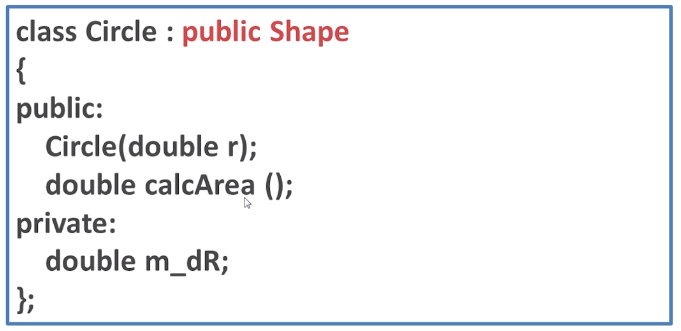




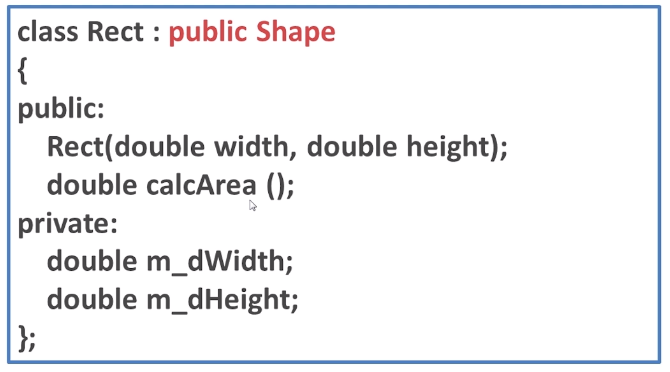
### 动态多态

动态多态又叫晚绑定，可以理解为对不同的对象下发相同的指令，但是却执行不同的操作，即在真正运行的时候确定执行的函数。动态多态的前提是必须以封装和继承为前提才可以。









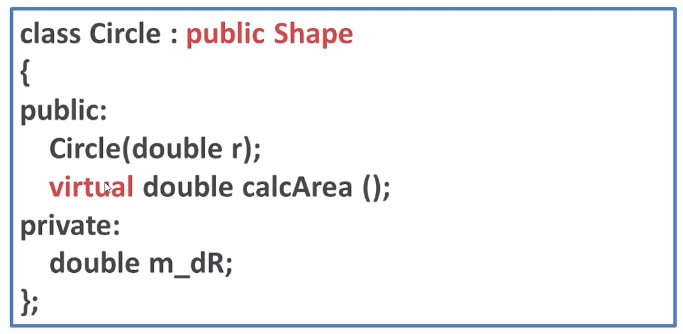


如果采用如下的调用方式，此时调用的都是父类的计算面积，即打印两行“cal area”，而不是调用子类具体的计算函数，这个结果不是我们想要的：

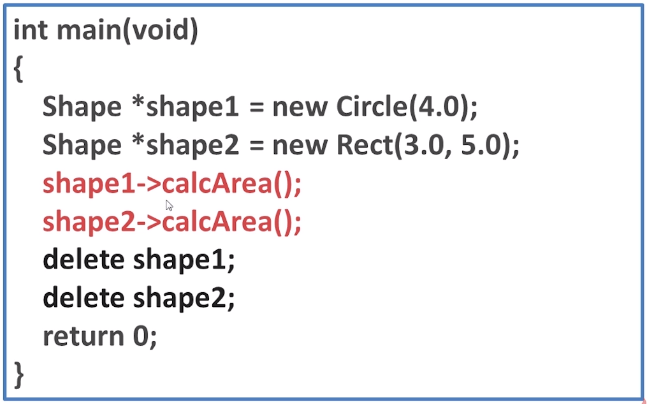


如果想要实现动态多态，需要使用virtual虚函数，对函数做如下修改：

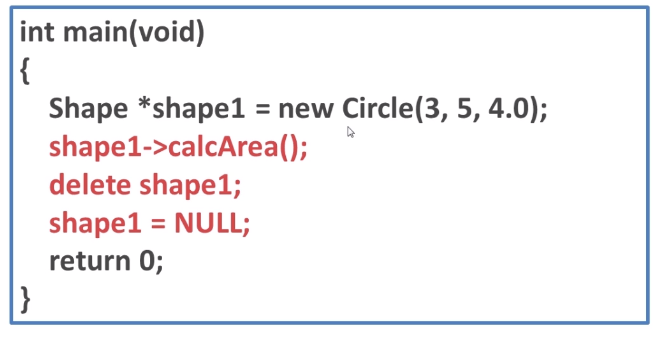




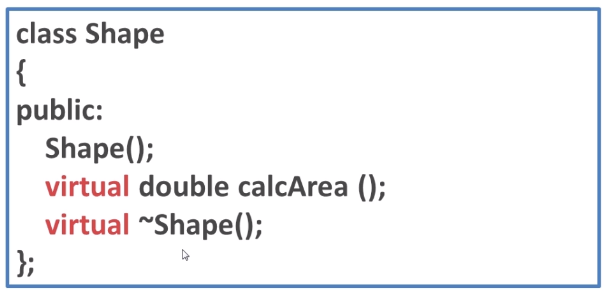
在定义子类的时候继承自父类的virtual函数加一个virtual，如果不加默认会添加virtual，但是还是推荐显式增加virtual关键字。

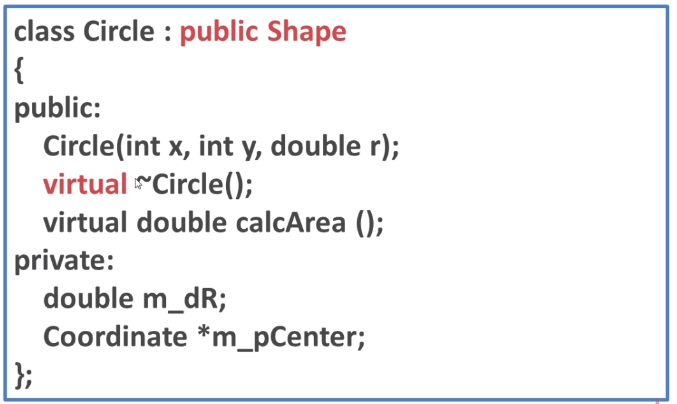


### 虚析构函数



在上面的例子中，调用虚函数是正确的，执行结果与预期一致，但是在delete内存释放的时候就会有问题，这里释放的是父类的指针，则调用的是父类的析构函数，并不会调用子类Circle的析构函数，那么就造成内存泄露了。因此引入了虚析构函数，即用virtual修饰析构函数，此时就不会造成内存泄露。





## 纯虚函数

## 抽象类