目录

[1. 函数重载的本质 2](#_Toc95830735)

[2. 默认函数简介 2](#_Toc95830736)

[3. extern “C”介绍 2](#_Toc95830737)

[4. #pragma once 简介 4](#_Toc95830738)

[5. 内联函数简介 4](#_Toc95830739)

[6. 引用的简介 5](#_Toc95830740)

[7. 对象的内存布局 6](#_Toc95830741)

[8. this指针的重点简介 7](#_Toc95830742)

[9. 内存空间的布局 9](#_Toc95830743)

[10. 构造函数 9](#_Toc95830744)

[11. 类的成员相关 12](#_Toc95830745)

[12. 析构函数 15](#_Toc95830746)

[13. 命名空间 16](#_Toc95830747)

[14. 继承以及相应的特点 16](#_Toc95830748)

[15. 多态以及虚函数 17](#_Toc95830749)

[16. 拷贝构造函数 19](#_Toc95830750)

[17. 隐式构造 20](#_Toc95830751)

[18. 友元 21](#_Toc95830752)

[19. 内部类 21](#_Toc95830753)

[20. 局部类 22](#_Toc95830754)

[21. 运算符重载 23](#_Toc95830755)

[22. 模板 26](#_Toc95830756)

[23. 类型转换 28](#_Toc95830757)

[24. Lambda表达式 30](#_Toc95830758)

[25. 异常 31](#_Toc95830759)

[26. 智能指针 34](#_Toc95830760)

### 函数重载的本质

**C++编译器默认会对符号名（比如函数名）进行改编、修饰（新的函数名会包含参数信息）**，有些地方翻译为“命名倾轧”。重载时会生成多个不同的函数名，不同编译器（MSVC、g++）有不同的生成规则。

通过IDA打开【VS\_Release\_禁止优化（防止编译器优化）】可以看到

### 默认函数简介

默认参数只能按照右到左的顺序

**如果函数同时有声明、实现，默认参数只能放在函数声明中**

默认参数的值可以是常量、全局符号（全局变量、函数名）

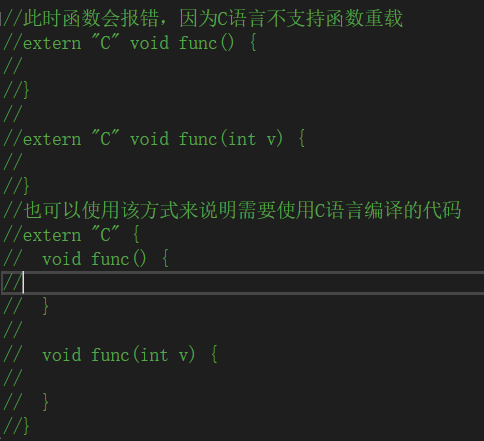
**注意：**函数重载、默认参数可能会产生冲突、二义性（建议优先选择使用默认参数）

**本质：函数调用的时候，首先会将省略的参数使用默认参数补全，然后再call对应的函数。即使用默认参数和使用相同参数补全的调用时生成的机器码是一致的。**

### extern “C”介绍

被extern "C"修饰的代码会按照C语言的方式去编译

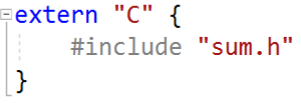
如果函数同时有声明和实现，要让**函数声明被extern "C"修饰，函数实现可以不修饰**



**用途：**

C++在**调用C语言API时**，需要使用extern "C"修饰C语言的函数声明

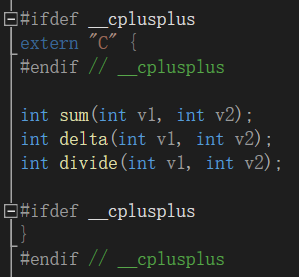
**原因：由于1中提到的的命名倾轧，所以c++代码与c代码最后汇编结果产生的汇编函数名是不一致的，导致无法正确call到需要的函数。**



**但该写法依旧不方便,若在 c库的sum.h中使用extern “C” 的话，将会导致项目文件中其他C文件无法使用该头文件（因为C语言中，没有extern “C”这种语法）。**

有时也会在编写C语言代码中直接使用extern “C” ，这样就可以直接被C++调用

使用宏 **\_\_cplusplus** 来确认当前编译环境是否为C++环境，所以讲第三方库的头文件写法如下所示，从而避免C++与C代码混用带来的问题。如下图所示。



### #pragma once 简介

◼ 我们经常使用#ifndef、#define、#endif来防止头文件的内容被重复包含

◼ #pragma once可以防止整个文件的内容被重复包含

◼ 区别

#ifndef、#define、#endif受C\C++标准的支持，不受编译器的任何限制

有些编译器不支持#pragma once（较老编译器不支持，如GCC 3.4版本之前），兼容性不够好

**#ifndef、#define、#endif可以针对一个文件中的部分代码**，而**#pragma once只能针对整个文件**

### 内联函数简介

◼ 使用inline修饰函数的声明或者实现，可以使其变成内联函数

建议**声明和实现都增加inline修饰**

◼ 特点

编译器**会将函数调用直接展开为函数体代码**

可以减少函数调用的开销

会增大代码体积

◼ 注意

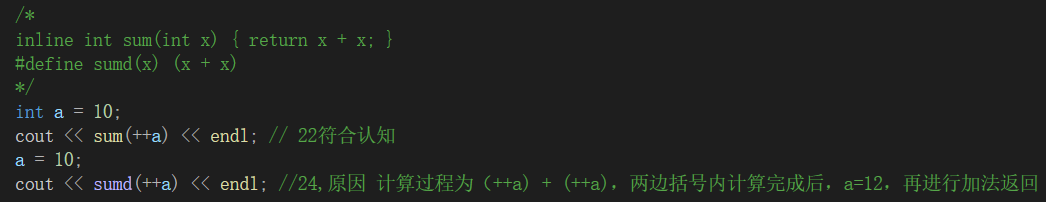
尽量不要内联超过10行代码的函数

有些函数即使声明为inline，也不一定会被编译器内联，比如**递归函数**

◼ 与宏的比较

内联函数和宏，都可以减少函数调用的开销

对比宏，内联函数多了语法检测和函数特性



### 引用的简介

◼ **注意点：**

1.引用相当于是变量的别名（基本数据类型、枚举、结构体、类、指针、数组等，都可以有引用）

2.对引用做计算，就是对引用所指向的变量做计算

3.在**定义**的时候就**必须初始化，一旦指向了某个变量，就不可以再改变，“从一而终”（原因也很简单，第一个是初始化，后面的相当于赋值，而直接修改了第一个变量的值）**

4.可以利用引用初始化另一个引用，相当于某个变量的多个别名

5.**不存在【引用的引用、指向引用的指针、引用数组】**

◼ 引用存在的价值之一：

比指针更安全、函数返回值可以被赋值

**本质：**

◼ **引用的本质就是指针**，只是编译器削弱了它的功能，所以引用就是弱化了的指针

◼ 一个引用占用一个指针的大小

**常引用：**

◼ 引用可以被const修饰，这样就无法通过引用修改数据了，可以称为常引用

**const必须写在&符号的左边，才能算是常引用**

◼ const引用的特点

可以指向**临时数据**（常量、表达式、函数返回值等）**（即非const引用只能指向左值，const可以指向右值）**

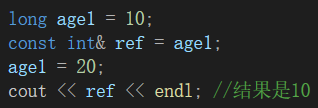
可以指向不同类型的数据

作为函数参数时（此规则也适用于const指针）

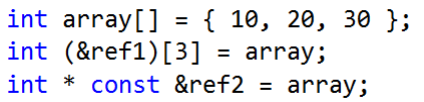
✓ 可以接受const和非const实参（**非const引用，只能接受非const实参**）

✓ **可以跟非const引用构成重载**

◼ 当常引用指向了不同类型的数据时，会产生临时变量，即**引用指向的并不是初始化时的那个变量，如下图所示：**



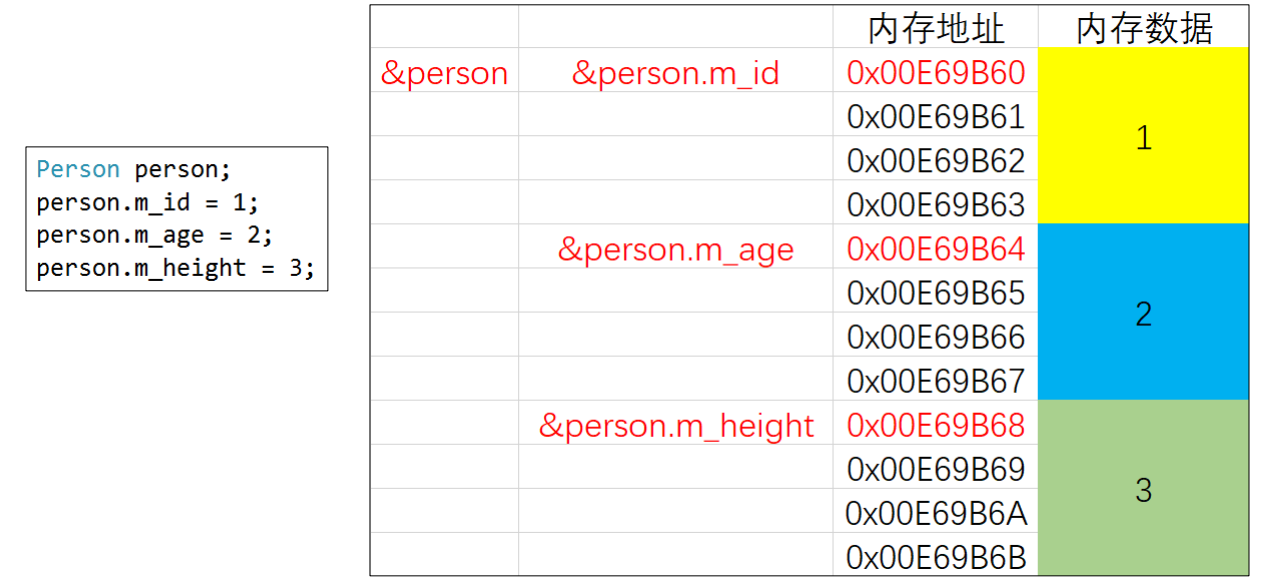
**数组的引用示例：（注意array完全可以看为int\*类型,但由于是常量，故要加上const）**



### 对象的内存布局

**注意不要忘记了对齐原则**





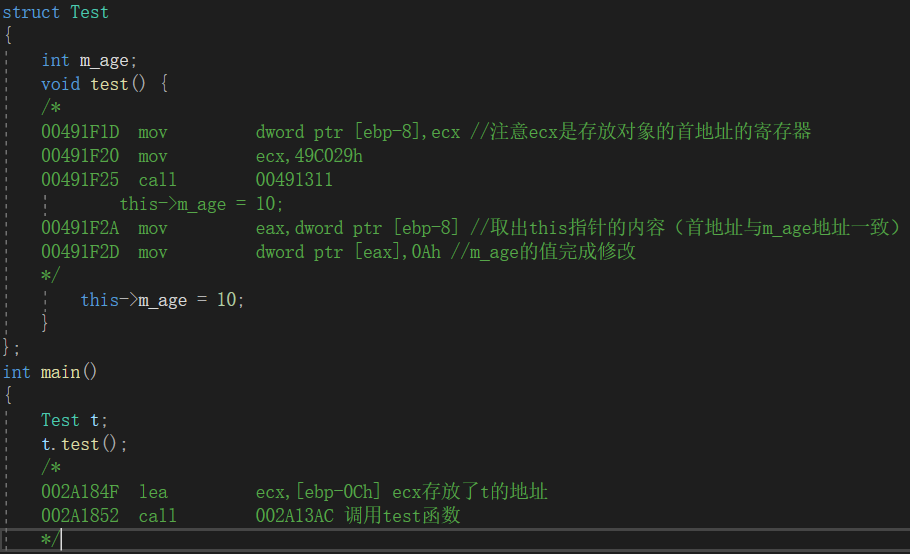
**注意：类的成员函数将会存储在代码区，只有成员数据会存储在栈区或者堆区。**

### this指针的重点简介

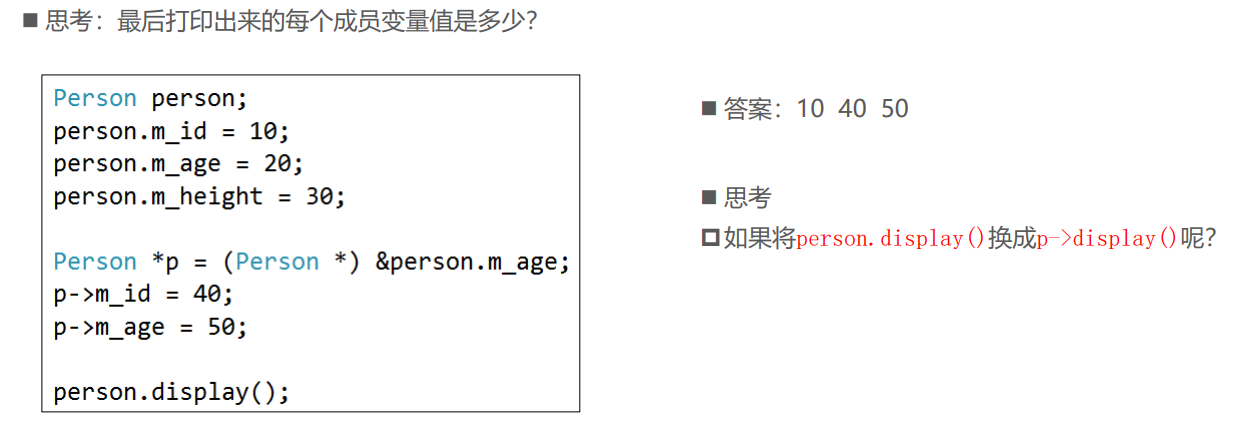
◼ this是指向当前对象的指针

◼ 对象在调用成员函数的时候，会自动传入当前对象的内存地址

在汇编中的时候可以看到的一些显著的特征，如下图所示



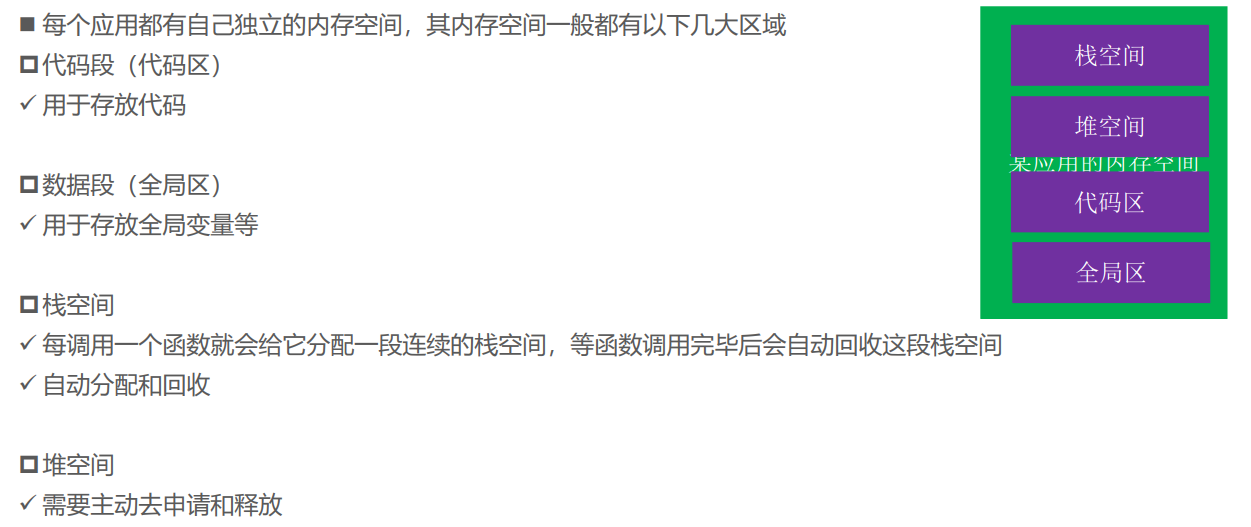
**思考：**



**原因：p存放的地址是 person的首地址+4。**

**结果：display的结果为 40,50，随机值。结果说明this指针访问成员变量的本质是通过“地址+偏移量”实现的，故虽然使用了成员变量的名字，然而访问的结果却如上所示。**

### 内存空间的布局



### 构造函数

◼ 构造函数（也叫构造器），在对象创建的时候自动调用，一般用于完成对象的初始化工作

◼ 特点

函数名与类同名，无返回值（void都不能写），可以有参数，可以重载，可以有多个构造函数

一旦自定义了构造函数，必须用其中一个自定义的构造函数来初始化对象

**◼ 注意**

**通过malloc分配的对象不会调用构造函数**

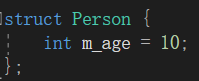
◼ 一个广为流传的、很多教程\书籍都推崇的错误结论：

**错误：**默认情况下，编译器会为每一个类生成空的无参的构造函数

**正确：**在某些特定的情况下，编译器才会为类生成空的无参的构造函数

特定情况：

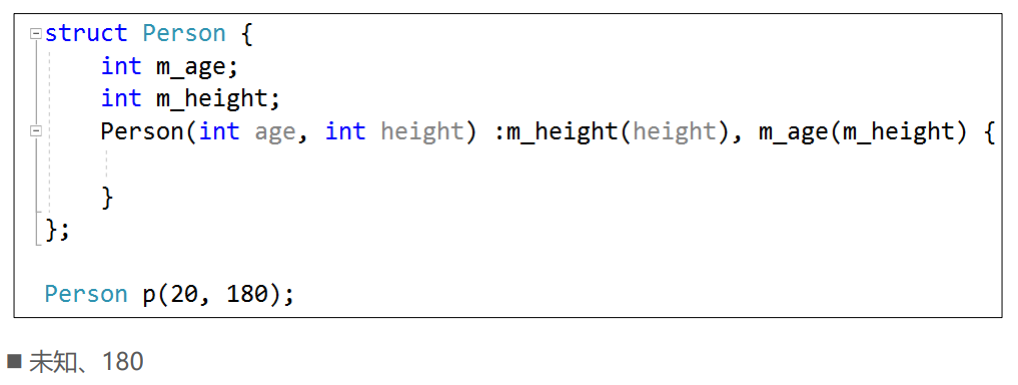
1. 类的成员变量在声明的时候就进行了初始化。



**初始化列表特点：**

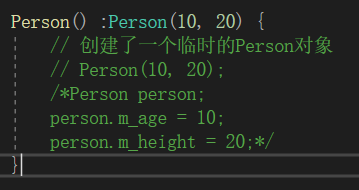
1.只能用在构造函数中

2.**初始化顺序只跟成员变量的声明顺序有关，故有下图所示的结果：**



**原因正是先初始化m\_age，故导致了未知的出现。**

3.构造函数**调用构造函数必须写在初始化列表中，如下图所示：**



**注意，如果函数声明和实现是分离的情况下，则：**

**初始化列表**只能写在函数的**实现**中

**默认参数**只能写在函数的**声明**中

**构造时调用父类的构造函数（前提是父类有构造函数）:**

◼ 子类的构造函数默认会调用**父类的无参**构造函数

◼ 如果子类的构造函数显式地调用了父类的有参构造函数，就不会再去默认调用父类的无参构造函数

◼ 如果**父类缺少无参构造函数，子类的构造函数必须显式调用父类的有参构造函数**

一个简单的示例如下所示：



**编译器自动生成的构造函数：**

◼ C++的编译器在某些特定的情况下，会给类自动生成无参的构造函数，比如

成员变量在**声明的同时进行了初始化**

有定义虚函数

虚继承了其他类

包含了对象类型的成员，且这个**成员有构造函数**（编译器生成或自定义）

父类有构造函数（编译器生成或自定义）

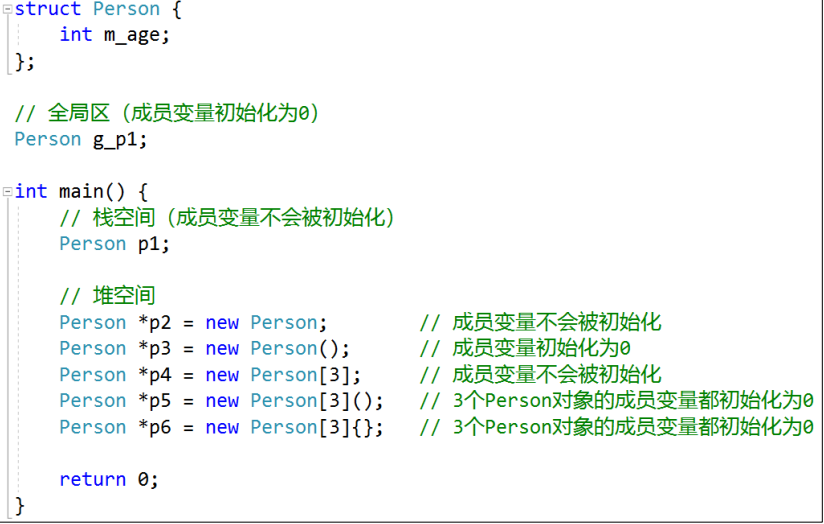
◼ 总结

对象创建后，**需要做一些额外操作时（比如内存操作、函数调用）**，编译器一般都会为其自动生成无参的构造函数

### 类的成员相关

**初始化：**

**默认情况下**



**自定义构造函数的情况下：**

**除了全局区，其他内存空间的成员变量默认都不会被初始化**，需要开发人员手动初始化。

**静态成员：**

◼ 静态成员：被**static**修饰的成员变量\函数

可以通过对象（对象.静态成员）、对象指针（对象指针->静态成员）、类访问（类名::静态成员）

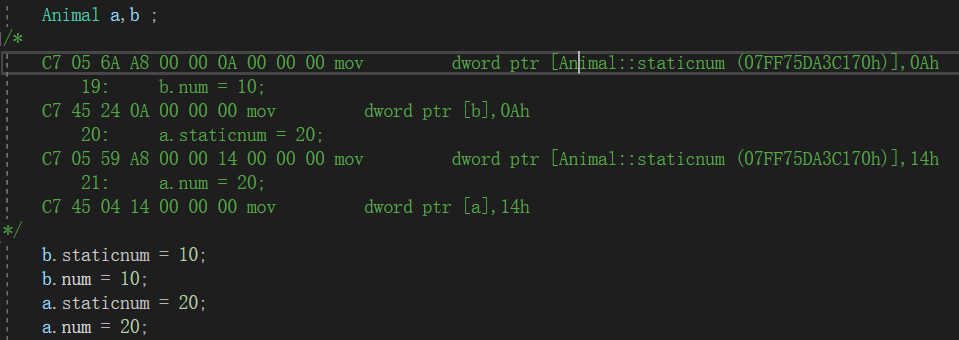
◼ 静态成员变量

存储在数据段（全局区，类似于全局变量），整个程序运行过程中只有一份内存

对比全局变量，它可以设定访问权限（**public、protected、private**），达到局部共享的目的

**必须初始化，必须在类外面初始化，初始化时不能带static**，如果类的声明和实现分离（在实现.cpp中初始化）

很明显的可以看出,**静态成员变量存储的地址是一个常量**，故其本质也可以说是一个**受限制的（public,protected,private访问限制）的全局变量**。



◼ 静态成员函数

**内部不能使用this指针**（this指针只能用在非静态成员函数内部）

不能是虚函数（虚函数只能是非静态成员函数）

**内部不能访问非静态成员变量\函数，只能访问静态成员变量\函数**

非静态成员函数内部可以访问静态成员变量\函数

构造函数、析构函数不能是静态

当声明和实现分离时，**实现部分不能带static**

**静态成员经典应用 – 单例模式：**

单例模式：设计模式的一种，保证某个类永远只创建一个对象

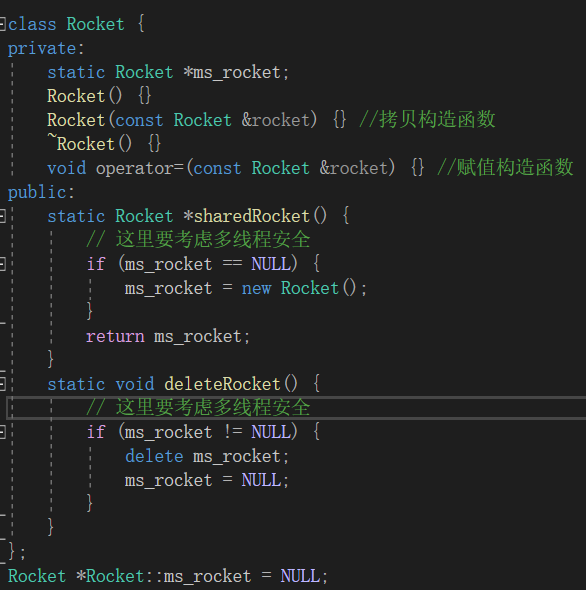
**要求：**

1.构造函数\析构函数私有化，拷贝构造函数和赋值构造函数也要将函数体设置为空。

2.定义一个私有的static成员变量指向唯一的那个单例对象

3.提供一个公共的访问单例对象的接口

4.**需要注意多线程安全的问题**



Const成员、引用成员：

◼ const成员：被const修饰的成员变量、**非静态成员函数（注意函数必须是非static）**

◼ const成员变量

**必须初始化（类内部初始化）**，可以在声明的时候直接初始化赋值

非static的const成员变量还可以在初始化列表中初始化

◼ const成员函数（非静态）

const关键字写在参数列表后面，**函数的声明和实现都必须带const**

**内部不能修改非static成员变量**

内部**只能调用const成员函数(继续保证内部不修改static成员变量)、static成员函数（static成员函数根本不能访问非static成员）**

非const成员函数可以调用const成员函数

**const成员函数和非const成员函数构成重载**

非const对象（指针）优先调用非const成员函数

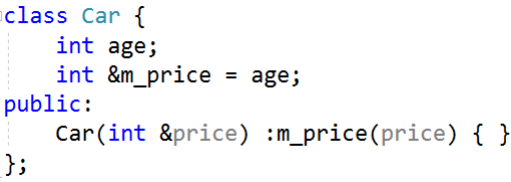
**const对象（指针）只能调用const成员函数、static成员函数**

**引用类型成员**

◼ 引用类型成员变量**必须初始化**（不考虑static情况）

在**声明的时候**直接初始化

通过**初始化列表**初始化



### 析构函数

◼ 析构函数（也叫析构器），在对象销毁的时候自动调用，一般用于完成对象的清理工作

◼ 特点

函数名以~开头，与类同名，无返回值（void都不能写），无参，不可以重载，有且只有一个析构函数

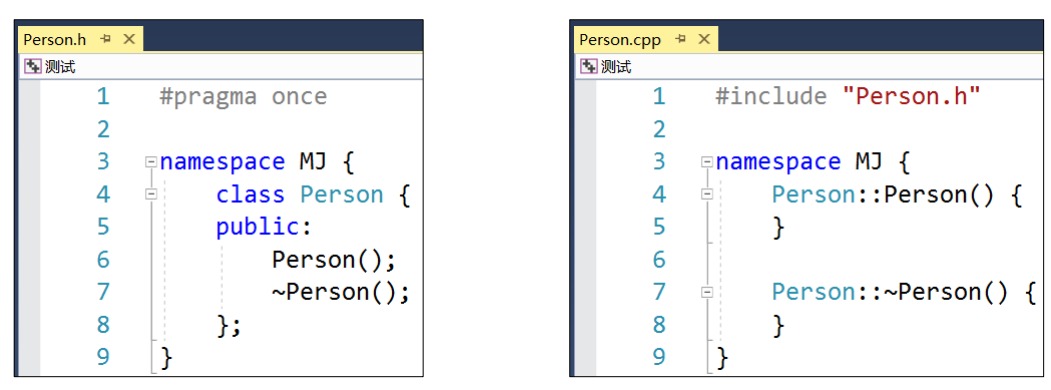
◼ 注意 通过malloc分配的对象free的时候不会调用析构函数

◼ 构造函数、析构函数要声明为public，才能被外界正常使用

**基类的析构函数必须是虚函数，因为delete的情况不会自动调用子类的析构函数。**

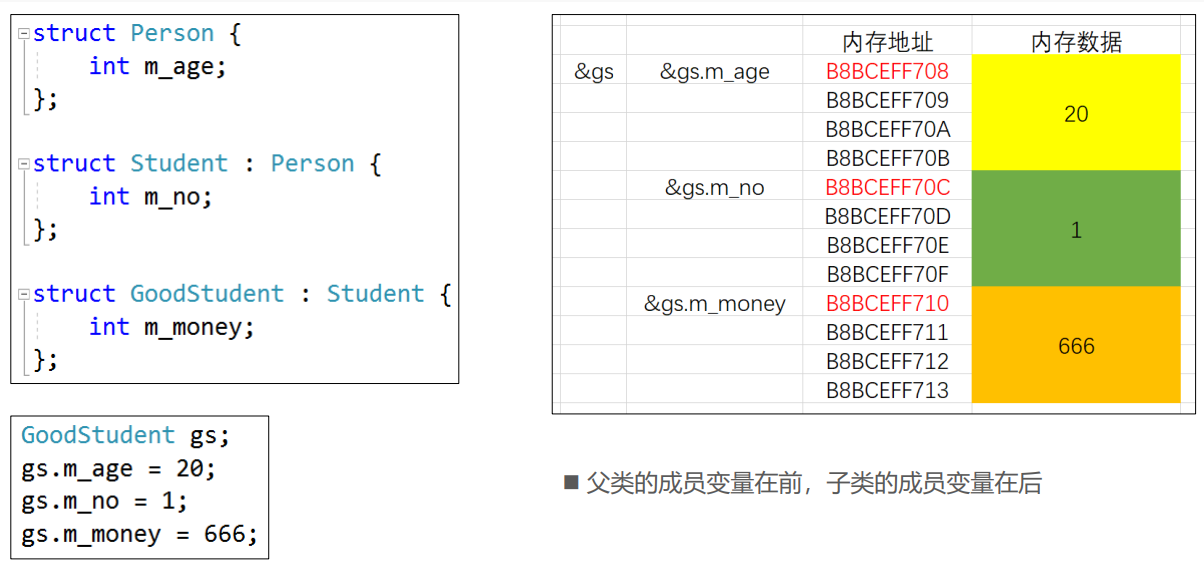
### 命名空间

命名空间可以用来避免命名冲突，注意命名空间在不同文件内也是可以合并的。



### 继承以及相应的特点

**对象的内存布局:**



**成员访问权限:**

◼ 成员访问权限、继承方式有3种

public：公共的，任何地方都可以访问（struct默认）

protected：子类内部、当前类内部可以访问

private：私有的，只有当前类内部可以访问（class默认）

◼ 子类内部访问父类成员的权限，是以下2项中权限最小的那个

成员本身的访问权限

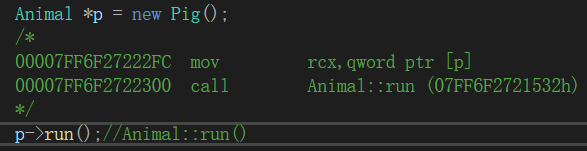
上一级父类的继承方式

◼ 开发中用的最多的继承方式是public，这样能保留父类原来的成员访问权限

◼ 访问权限不影响对象的内存布局

### 多态以及虚函数

◼ 默认情况下，编译器只会根据指针类型调用对应的函数，不存在多态**（即不使用virtual的情况）：**



可以很清楚的看出，**左侧的类型将决定函数的调用选择**。

◼ 多态是面向对象非常重要的一个特性

同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果

在运行时，可以识别出真正的对象类型，调用对应子类中的函数

◼ 多态的要素

子类**重写**父类的成员函数（override）

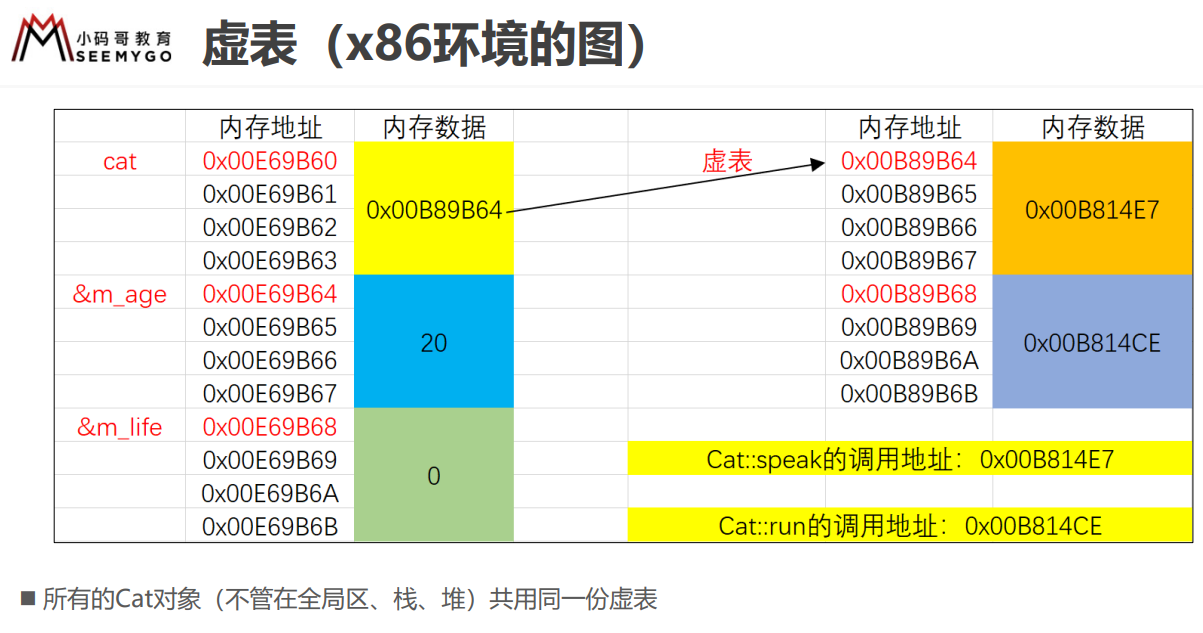
父类指针指向子类对象

利用父类指针调用重写的成员函数

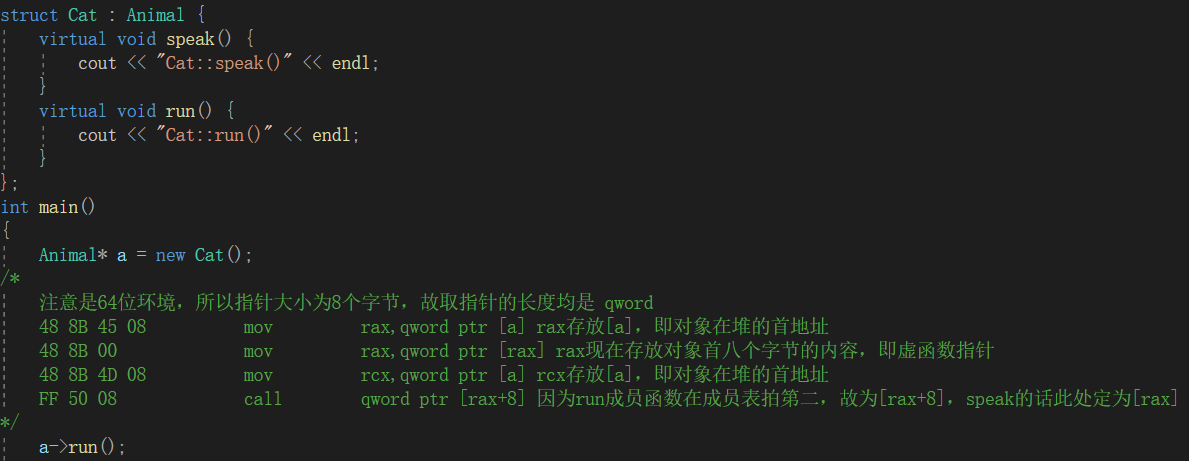
**虚函数:**

一旦成员函数含有虚函数，则**编译器会自动生成一个虚函数指针的成员变量。**

**虚函数使用的确定由指针初始化时等号右边的内容决定。**



一个虚函数调用的汇编分析，如下图所示：



**纯虚函数:**

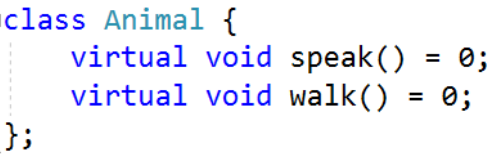
◼ 纯虚函数：没有函数体且初始化为0的虚函数，用来定义接口规范

◼ 抽象类（Abstract Class）

含有纯虚函数的类，不可以实例化（不可以创建对象）

抽象类也可以包含非纯虚函数、成员变量

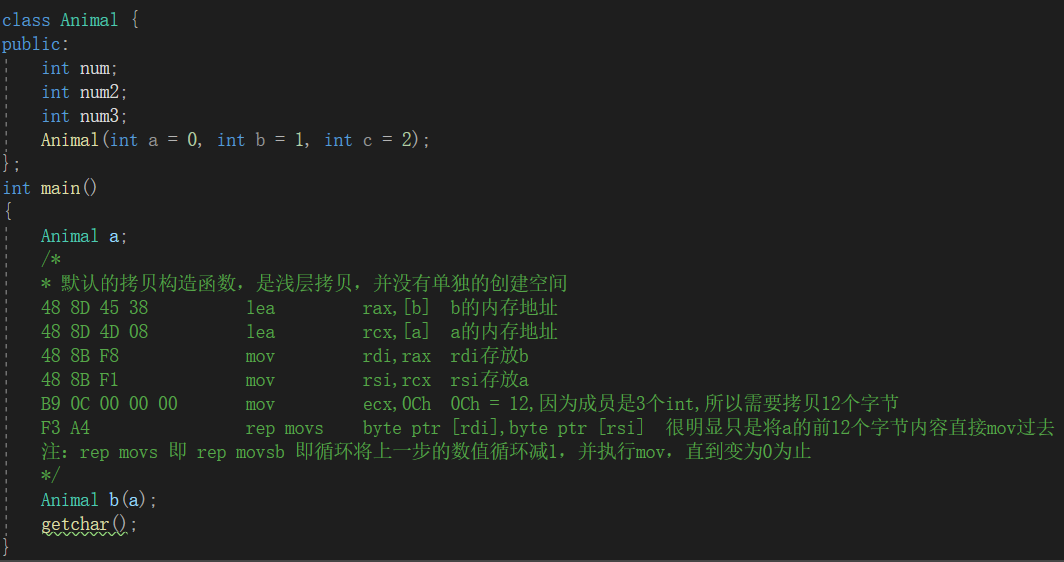
如果父类是抽象类，**子类没有完全重写纯虚函数，那么这个子类依然是抽象类**



### 拷贝构造函数

**编译器默认的拷贝构造函数**

只是简单的浅拷贝，如下图所示。

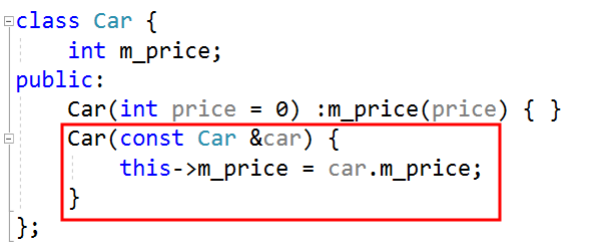


**自定义拷贝构造函数**

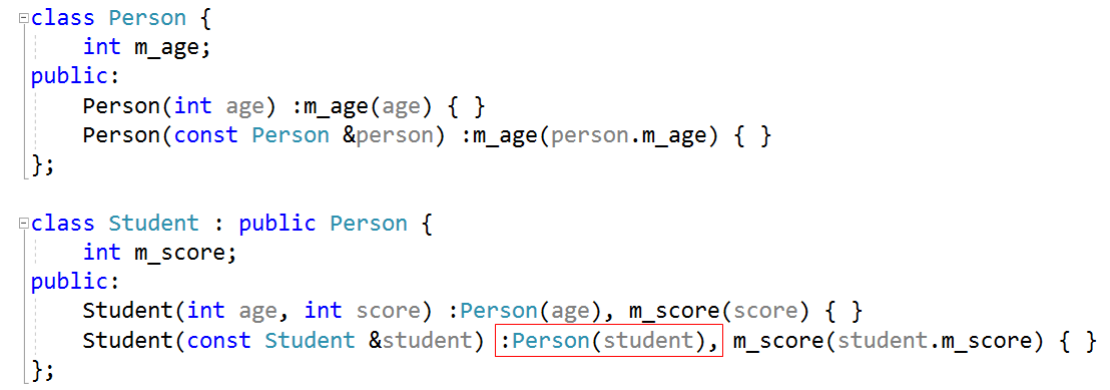
◼ 拷贝构造函数是构造函数的一种

◼ 当利用已存在的对象创建一个新对象时（类似于拷贝），就会调用新对象的拷贝构造函数进行初始化

◼ 拷贝构造函数的格式是固定的，接收一个const引用作为参数



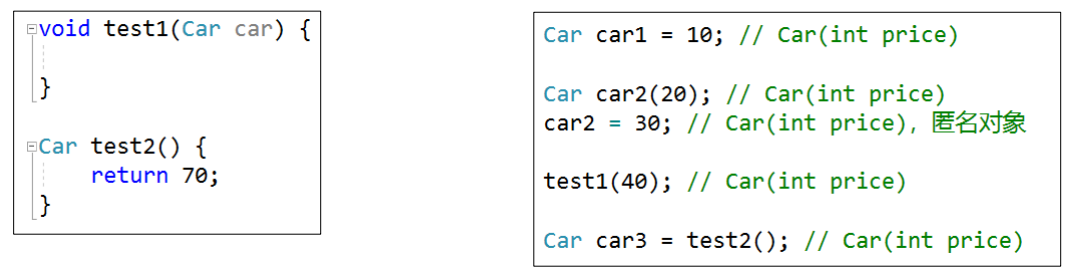
**调用父类的拷贝构造函数（初始化父类成员变量）**



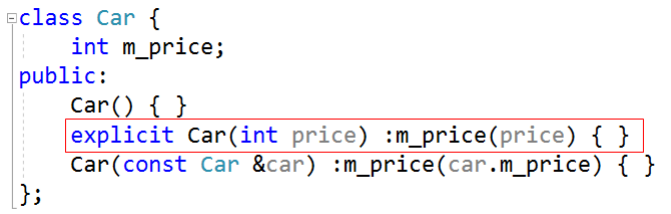
**注意：**如果子类没有显式调用父类的拷贝构造函数，编译器会主动调用父类无参的构造函数

### 隐式构造

◼ C++中存在隐式构造的现象：某些情况下，**会隐式调用单参数的构造函数**



◼ 可以通过关键字**explicit**禁止掉隐式构造



### 友元

◼ 友元包括友元函数和友元类

◼ 如果将函数A（**非成员函数**）声明为类C的友元函数，那么函数A就能直接访问类C对象的所有成员

◼ 如果将**类A**声明为类C的友元类，那么类A的所有成员函数都能直接访问类C对象的所有成员

◼ 友元破坏了面向对象的封装性，但在某些频繁访问成员变量的地方可以提高性能

### 内部类

◼ 如果将类A定义在类C的内部，那么类A就是一个内部类（嵌套类）

◼ 内部类的特点

支持public、protected、private权限

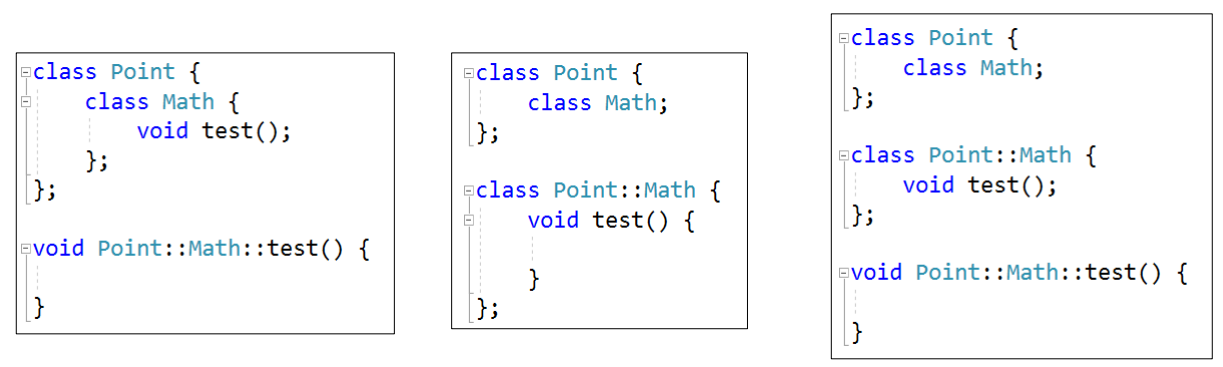
成员函数可以直接访问其外部类对象的所有成员（反过来则不行）

**成员函数可以直接不带类名、对象名访问其外部类的static成员**

不会影响外部类的内存布局

可以在外部类内部声明，在外部类外面进行定义

**声明和实现分离的示例**



### 局部类

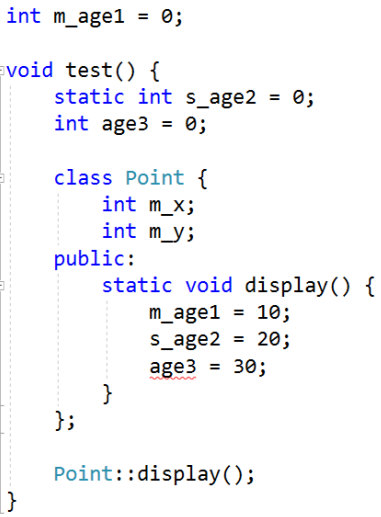
◼ 在一个**函数内部定义的类**，称为局部类

◼ 局部类的特点

作用域仅限于所在的函数内部

其所有的成员必须定义在类内部，不允许定义static成员变量

**成员函数不能直接访问函数的局部变量**（static变量除外）

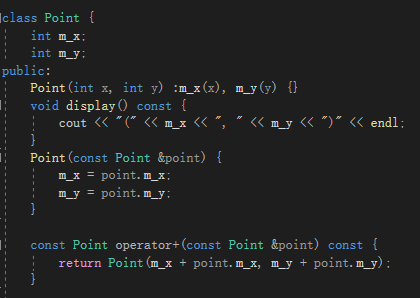


### 运算符重载

◼ 运算符重载（操作符重载）：可以为运算符增加一些新的功能

◼ 全局函数(使用**友元函数来访问私有变量**)、成员函数（**直接重载**）都支持运算符重载

**成员函数：**



**注意，函数两个const的使用原因：**

**第一个const:**

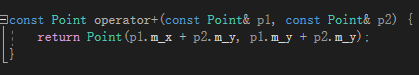
保证了返回的Point 对象是一个const 对象（右值），从而**保证返回的结果避免了被赋值的问题**，即 （p1+p2） = p3,这样不合理的代码。

**第二个const：**

**保证了连续运算**，因为（p1+p2）的返回结果是常量对象，故其只能调用常量函数，所以必须加第二个const,才能实现连续预算，即 p4 = p1 + p2 + p3;

**全局函数：**



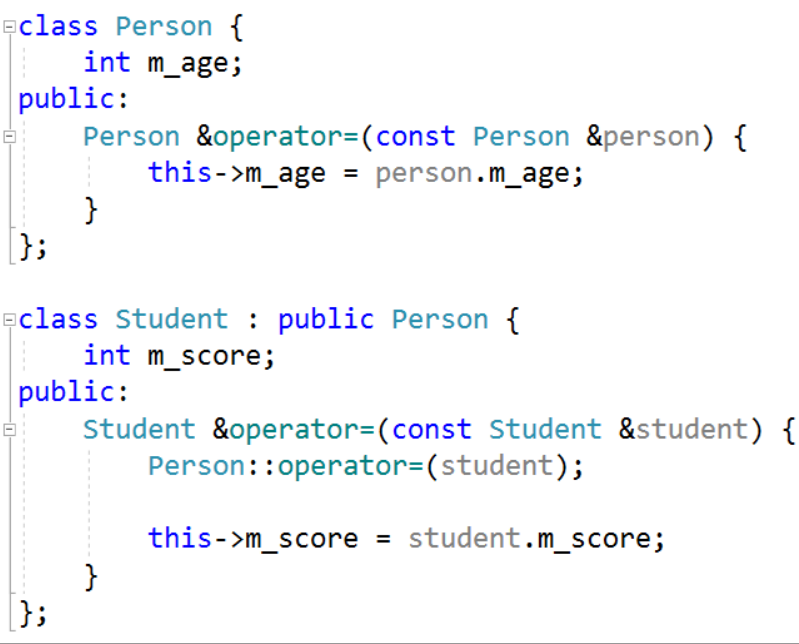


友元函数只需要一个const,因为其根本不属于类，所以没有常量成员函数的说法。

**调用父类的运算符重载函数：**

注意两点：

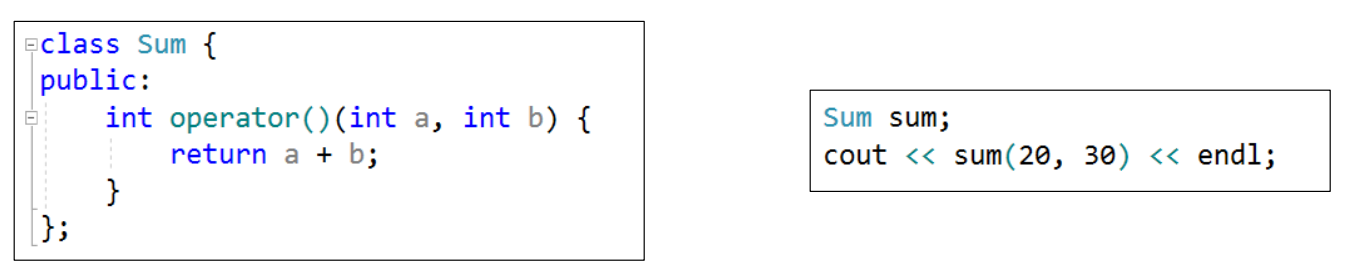
1. 父类也要重载该运算符
2. 子类重载运算符函数内，使用类作用域主导调用。



**仿函数：**

**本质：重载()运算符**

◼ 仿函数：将一个对象当作一个函数一样来使用



◼ 对比普通函数，它作为对象可以保存状态

**重载运算符其他注意点：**

◼ 有些运算符**不可以被重载**，比如

对象成员访问运算符：.

域运算符：::

三目运算符：?:

sizeof

◼ 有些运算符**只能重载为成员函数**，比如

赋值运算符：=

下标运算符：[ ]

函数运算符：( )

指针访问成员：->

### 模板

本质：编译器根据不同的类型，实例化相应的函数。

◼ 泛型，是一种将类型参数化以达到代码复用的技术，C++中使用模板来实现泛型

◼ 模板的使用格式如下

template <typename/class T>

typename和class是等价的

**◼** **模板没有（在本文件内）被使用时，是不会被实例化出来的**

◼ 一般将模板的声明和实现统一放到一个**.hpp**文件中

**◼ 模板的声明和实现如果分离到.h和.cpp中，会导致链接错误**

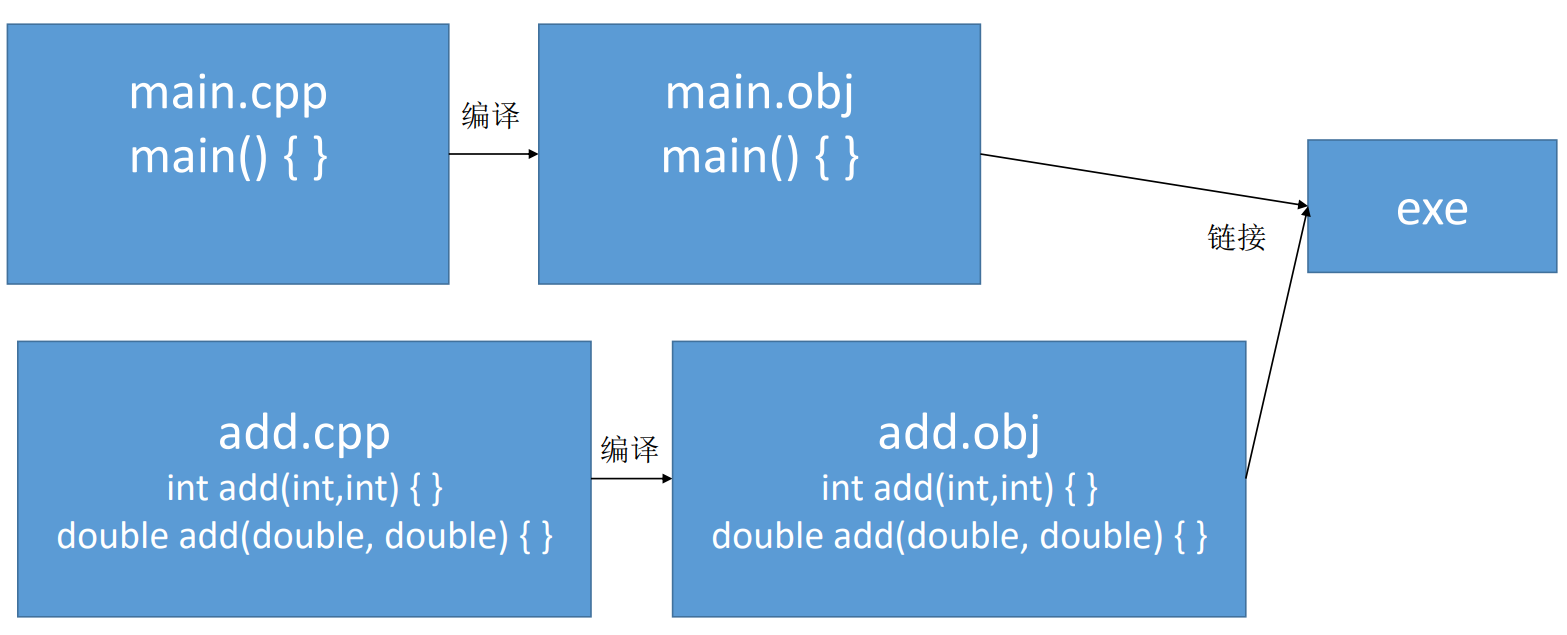
**原因分析：**

**背景知识：**

1. 模板没有（**在本文件内**）被使用时，是不会被实例化出来的。
2. 函数的声明是不存在地址的，函数的实现才是有地址的
3. 编译器在编译文件的过程中，**每一个文件是分开单独编译的**。每个文件中函数只有声明没有实现也是可以通过编译的，但**（只靠函数声明）生成的call的地址是不正确，在编译完后的链接阶段进行修正，替换为正确的函数实体地址。**

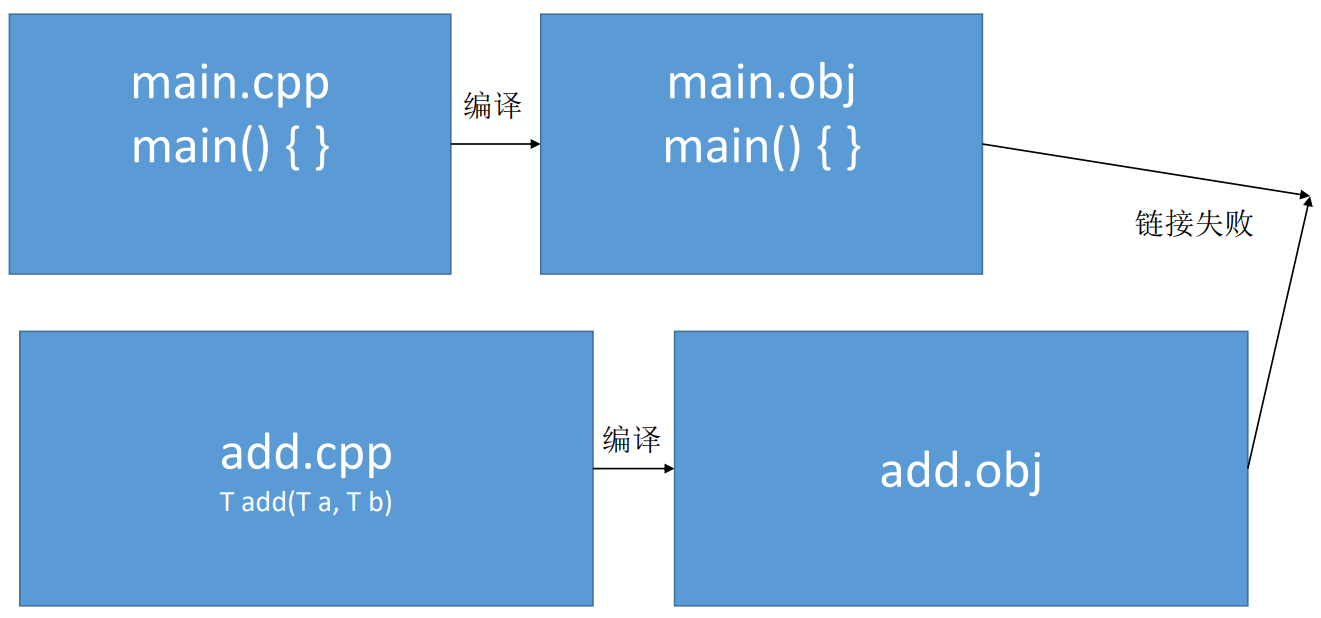
**分析：**

1. 一般情况下：



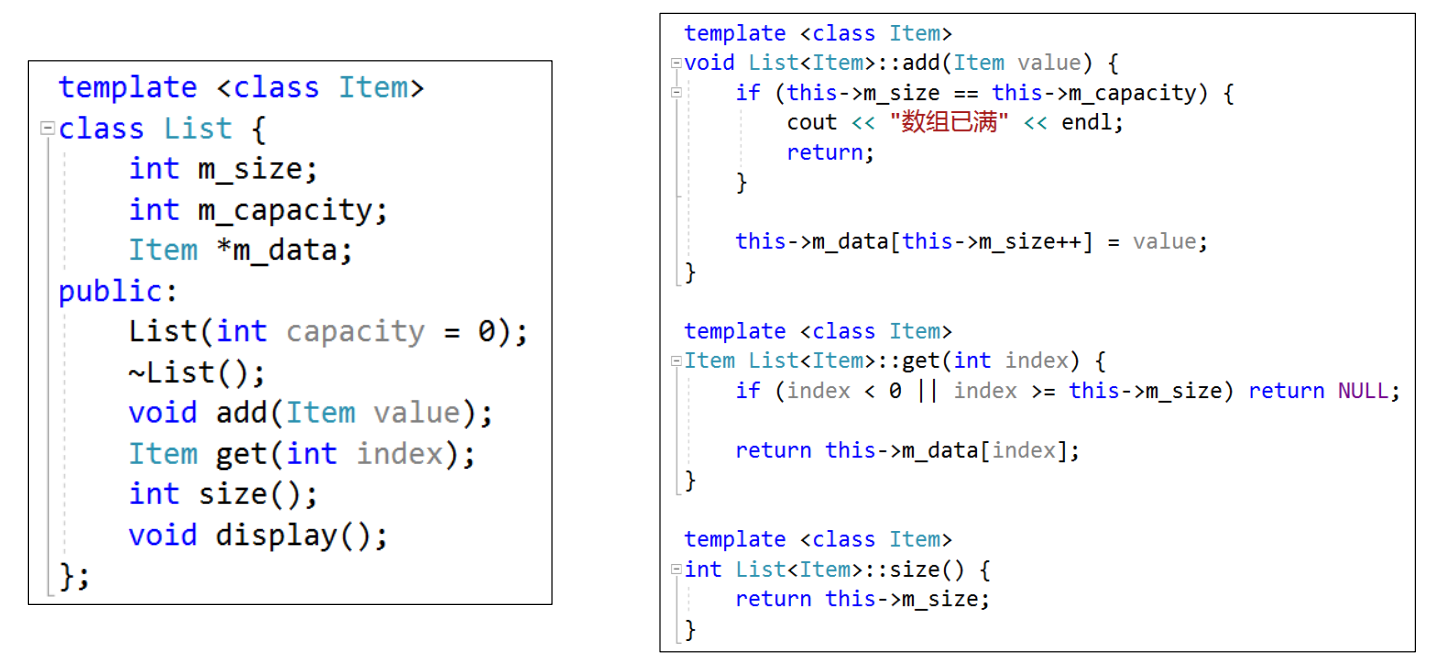
add.cpp中含有函数add的具体实现，故在链接阶段，可以将main.obj内的call add的地址替换为为正确的函数地址。

1. 模板函数声明和实现分离的情况下：

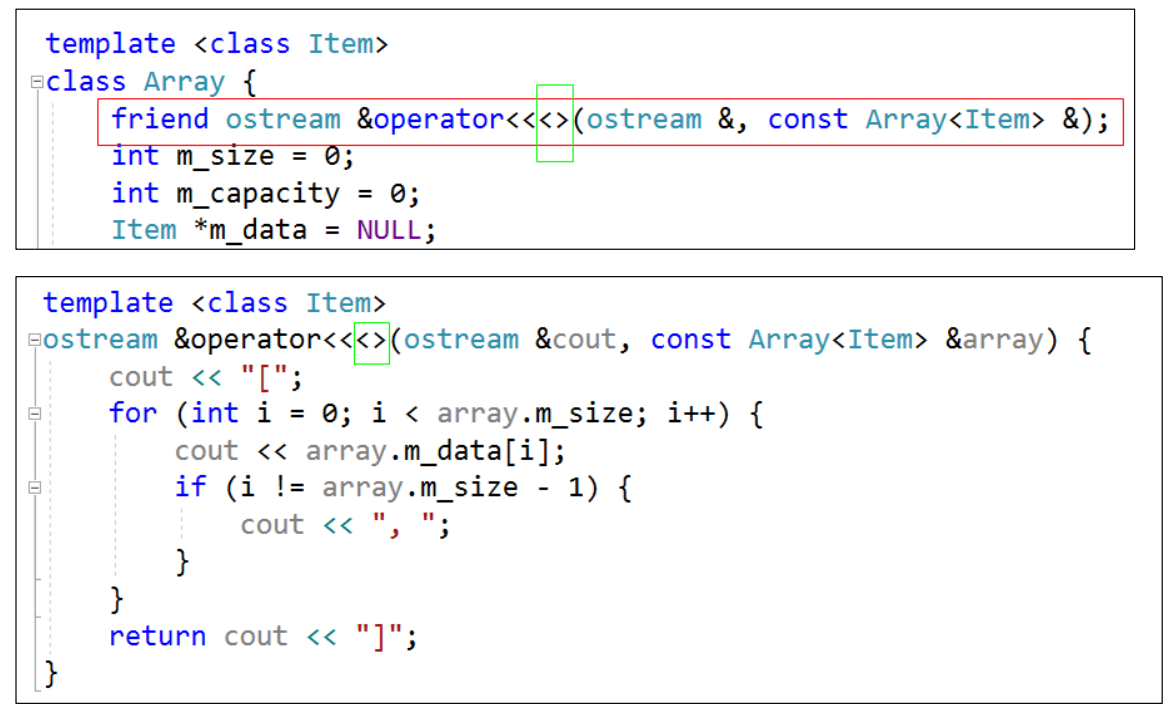


由于**在add.cpp内不存在模板函数的调用，所以并没有相应的具体函数实现，也就没有正确的函数地址。**所以连接阶段main.obj内的call add函数的地址均是不正确的，无法找到正确的地址进行替换，最终导致链接失败。

**类模板示例：**



注意，类模板使用友元函数的使用，需要在参数列表前添加 **<>,**如下如所示**。**



### 类型转换

◼ C语言风格的类型转换符

(type)expression

type(expression)

◼ C++中有4个类型转换符

1. static\_cast**:**

◼ 对比dynamic\_cast，缺乏运行时安全检测

◼ **不能交叉转换（不是同一继承体系的，无法转换）**

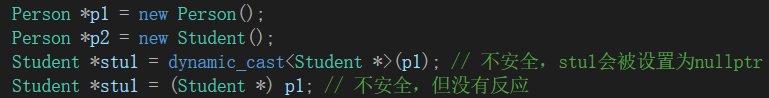
◼ 常用于**基本数据类型的转换、非const转成const**

◼ 使用范围较广



1. **dynamic\_cast**

**◼** 一般用于**多态类型的转换（没有虚函数的时候不存在多态）**，有运行时安全检测



1. **reinterpret\_cast：**

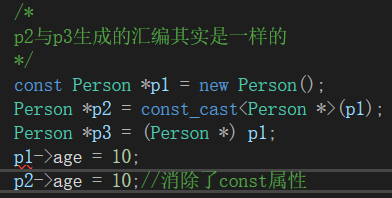
**◼ 属于比较底层的强制转换，没有任何类型检查和格式转换，仅仅是简单的二进制数据拷贝**

**◼ 可以交叉转换**

**◼ 可以将指针和整数互相转换**

1. const\_cast：

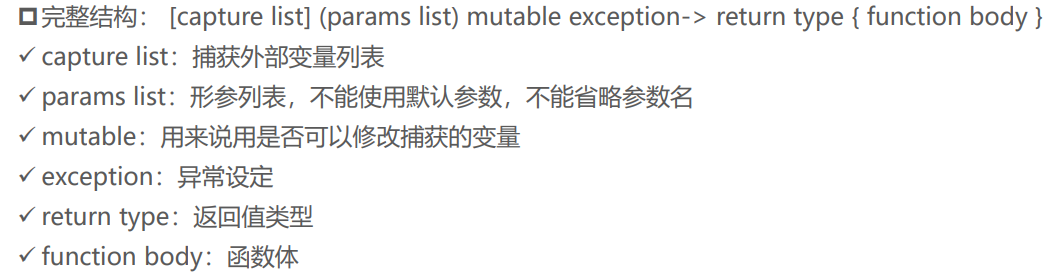
◼ 一般用于去除const属性，将const转换成非const



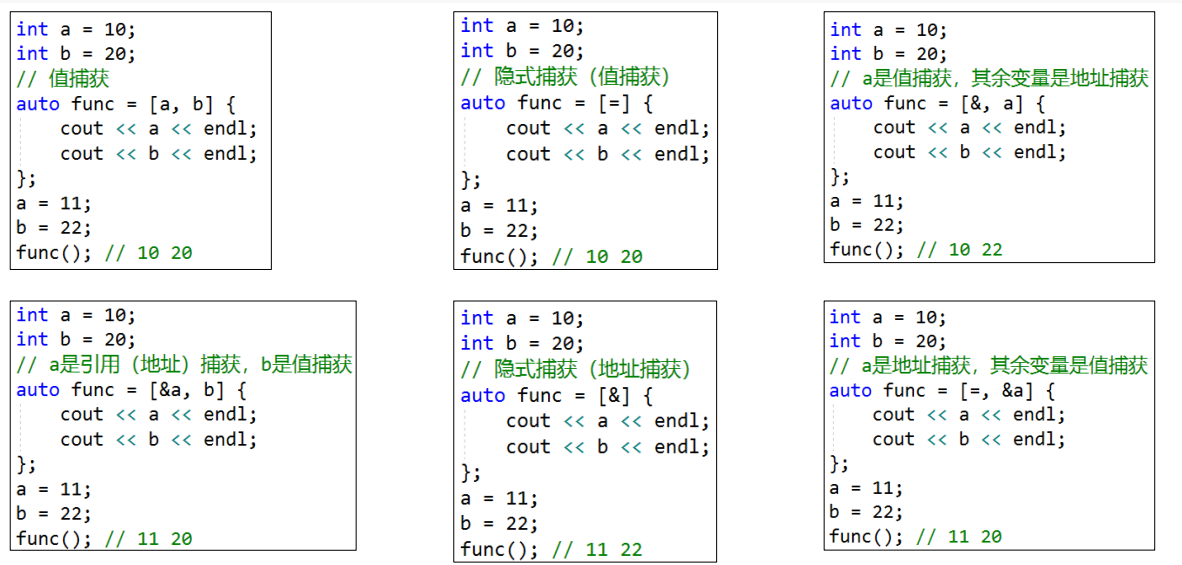
使用格式：xx\_cast(expression)

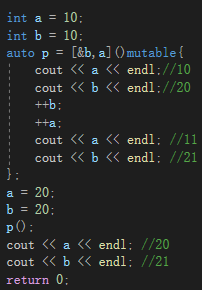
### Lambda表达式

**本质就是函数**



**Lambda的捕获和mutable示例**





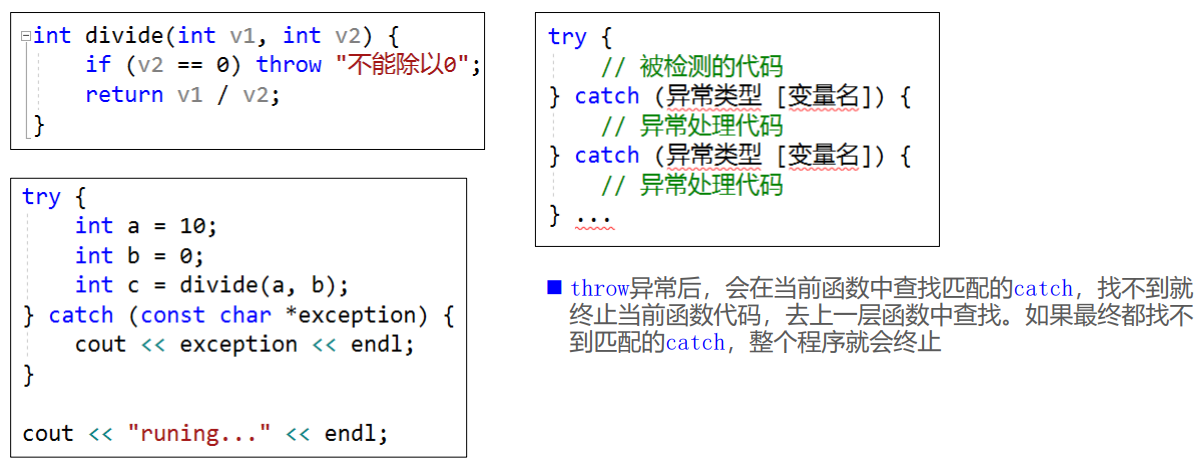
**值捕获：在Lambda函数定义**的时候，值便已经被确定，mutable虽然可以修改值，但对外界无影响，可视为在Lambda函数体内定义了一个可以修改的变量。

**地址捕获:**在**Lambda函数被调用**的时候才确定值，且mutable修改的结果会对外界的值进行修改。

### 异常

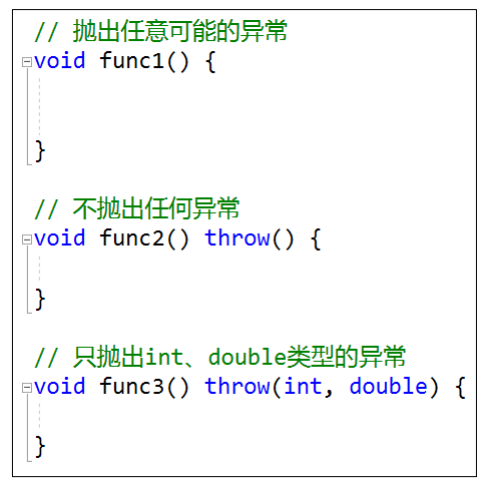
◼ 异常是一种在程序运行过程中可能会发生的错误（比如内存不够）

◼ 异常没有被处理，会导致程序终止



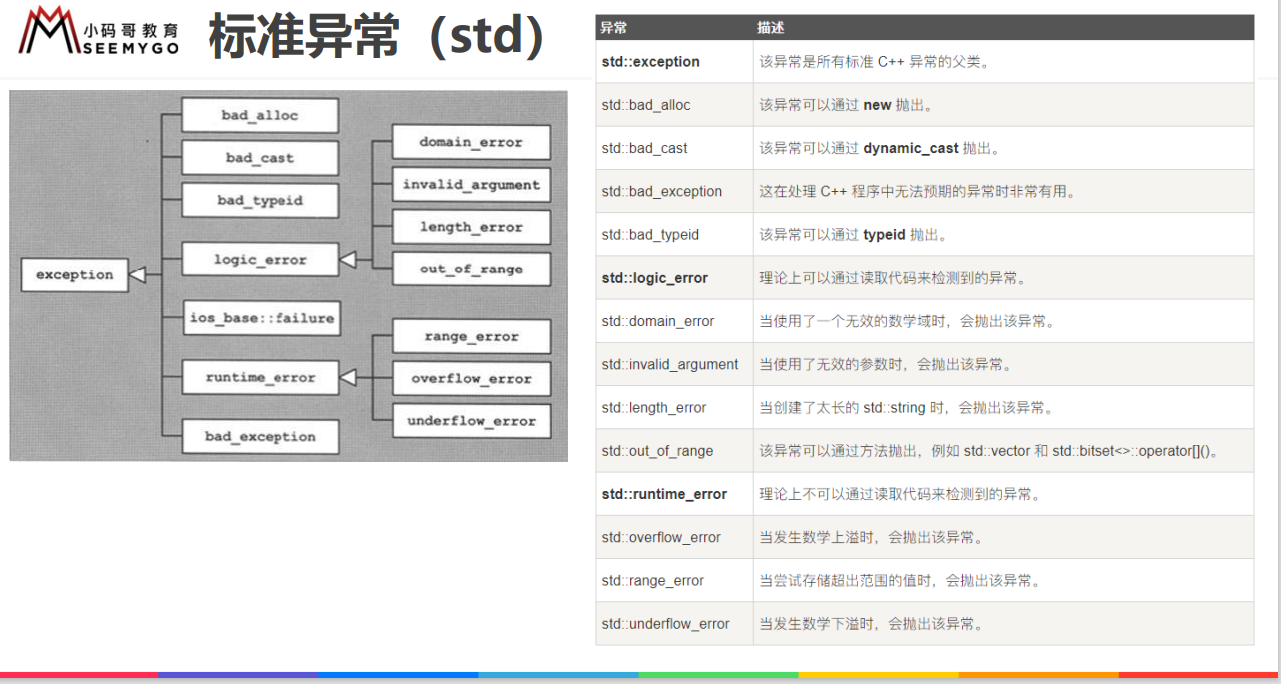
**异常的抛出声明：**

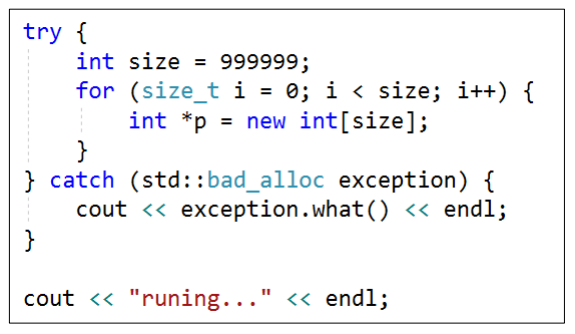
◼ 为了增强可读性和方便团队协作，如果函数内部可能会抛出异常，建议函数声明一下异常类型



**标准和自定义异常类型**

**标准异常(std):**

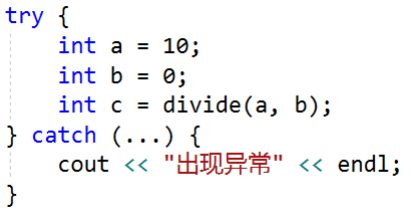




当然，也可以自定义异常



**拦截所有类型的异常：**



### 智能指针

**◼传统指针存在的问题**

需要手动管理内存

容易发生内存泄露（忘记释放、出现异常等）

释放之后产生野指针

**◼智能指针就是为了解决传统指针存在的问题**

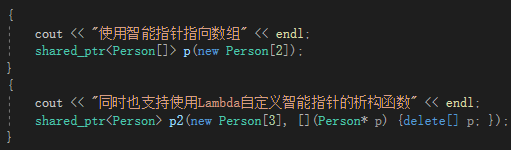
**注意：**智能指针自动回收空间的设计，故一定**只能指向堆空间的对象**（否则系统和智能指针两次回收资源，会带来异常）。

**auto\_ptr：**属于C++98标准

**在C++11中已经不推荐使用**（有缺陷，比如**不能用于数组**）

可以声明以及定义，但是由于其写死了 delete ,故不支持delete[ ]，从而导致不能用于数组。

**shared\_ptr：**属于C++11标准



◼ 一个shared\_ptr会对一个对象产生**强引用**（strong reference）

◼ 每个对象都有个与之对应的强引用计数，记录着当前对象被多少个shared\_ptr强引用着

可以通过shared\_ptr的**use\_count函数获得强引用计数**

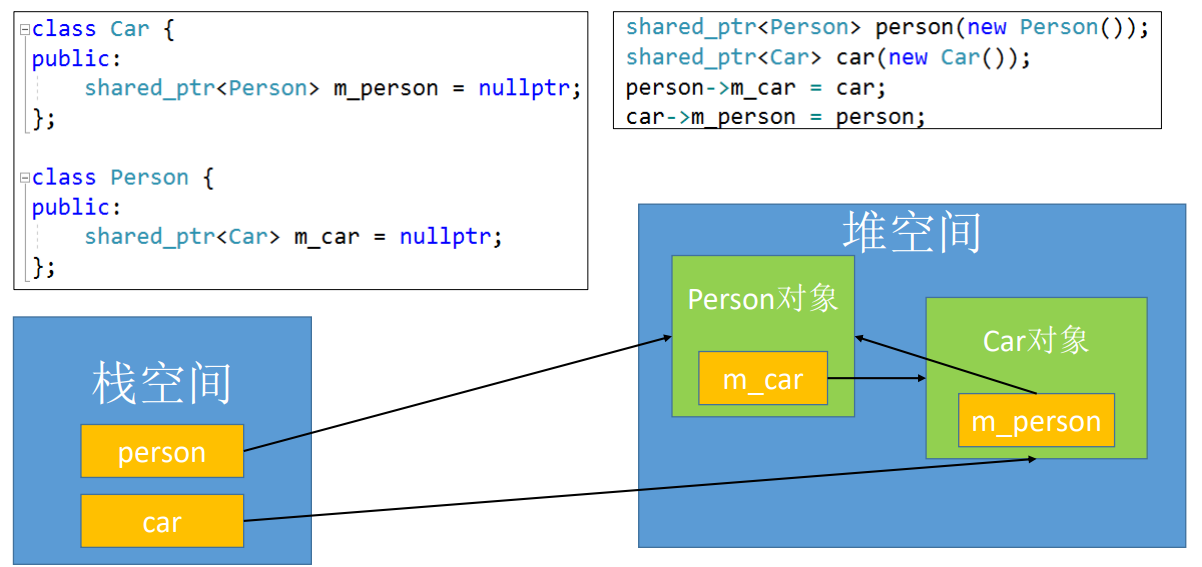
◼ 当有一个新的shared\_ptr指向对象时，对象的强引用计数就会+1

◼ 当有一个shared\_ptr销毁时（比如作用域结束），对象的强引用计数就会-1

◼ 当一个对象的强引用计数为0时（没有任何shared\_ptr指向对象时），对象就会自动销毁（析构）

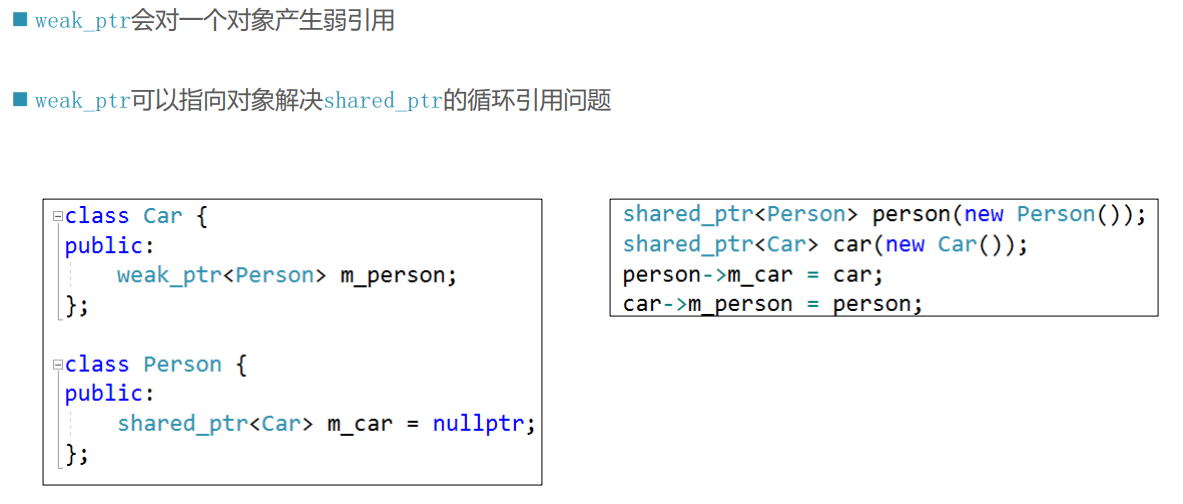
**注意：**

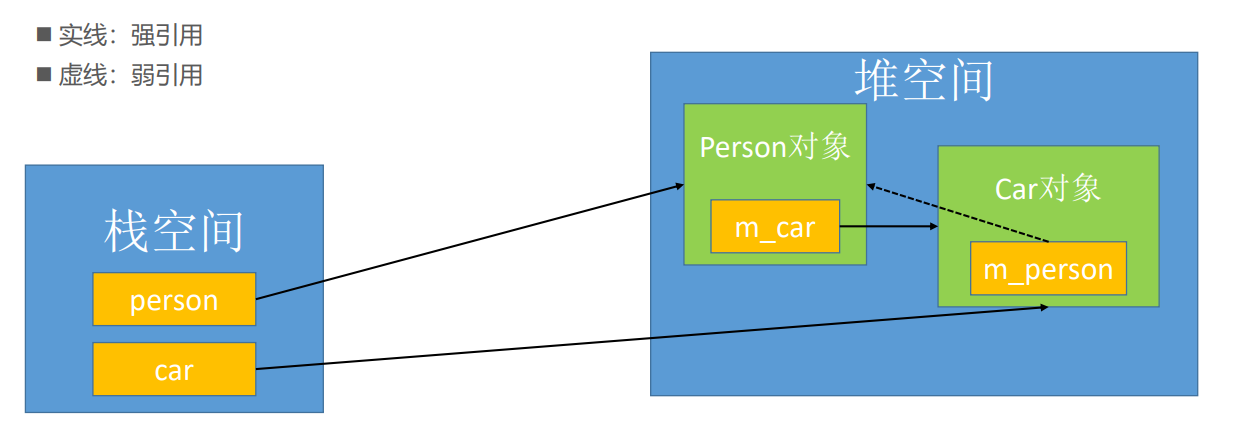
1. **智能指针不应与普通指针混用；C++ Primer P413**
2. **不要使用get来为另一个智能指针初始化或者赋值； C++ Primer P414**
3. **shared\_ptr的循环引用：**



**智能指针person和car准备释放空间的时候，由于两者的成员变量互相强引用，导致计数均为1，导致两个堆空间均不能释放。**

**解决方法：**





**unique\_ptr：**属于C++11标准

◼ unique\_ptr也会对一个对象产生强引用，它可以**确保同一时间只有1个指针指向对象**

◼ unique\_ptr销毁时（作用域结束时），其指向的对象也就自动销毁了

◼可以使用**std::move函数转移unique\_ptr的所有权**

