

一、C++

**1、static关键字作用**

（1）全局静态变量

- 在全局变量前面加上static关键字，全局变量就定义成一个全局静态变量；

- 数据存放在静态存储区，在整个程序运行期间一直存在；

- 初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显示初始化）；

- 作用域：全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾；

（2）局部静态变量

- 在局部变量前加上关键字static，局部变量就成为一个局部静态变量；

- 数据存放在静态存储区；

- 初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显示初始化），只会被初始化一次，后面的初始化都不会再实行；

- 作用域：其生命周期是全局的，在整个源文件执行完毕后释放，而不会在某个当前函数执行完就释放；

（3）静态函数

- 在函数返回类型前加static，函数就定义为静态函数。函数的定义和声明在默认情况下都是extern的，但静态函数只是在声明他的文件中可见，不能被其他文件所用；

- 在函数面前加static，不会与其他cpp文件的同名函数引起冲突；

（4）类的静态成员

- 在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类所有对象共享的成员，不属于某个类。对多个对象来说，静态数据成员只存储一处，供所有对象使用；

（5）类的静态函数

- 静态成员函数和静态成员一样，都不是对象成员。因此，对静态成员的引用不需要用对象名。在静态成员函数的实现中不能直接引用类中说明的非静态成员（若引用，需要通过对象来引用），可以引用类中说明的静态成员

**2、C++与C的区别**

（1）C是面向过程的语言，C++是面向对象的语言；

（2）C++具有封装、继承、多态三种特性；

（3）C和C++动态管理内存的方法不一样，C是使用malloc/free函数，而C++除此之外还有new/delete关键字；

（4）C++中有引用，C中没有；

（5）C++中的类是C没有的，但是C中的struct是可以在C++中正常使用，并且C++对struct类中的默认继承权限是公有的，而对class类默认继承权限是私有的；C语言struct中是没有函数体的，但可以有函数指针。

（6）C++支持函数重载，C不支持；

**3、C++中四种强制转换**

（1）static\_cast static\_cast<type-id> (expression)

主要用于基本数据类型之间的转换。该运算符把 expression 转换为 type-id 类型，但没有运行时类型检查来保证转换的安全性。

主要用法如下：(如**int** y **=** **static\_cast<int>**(x))

    - 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。

    进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；

    进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查， 所以是不安全的。

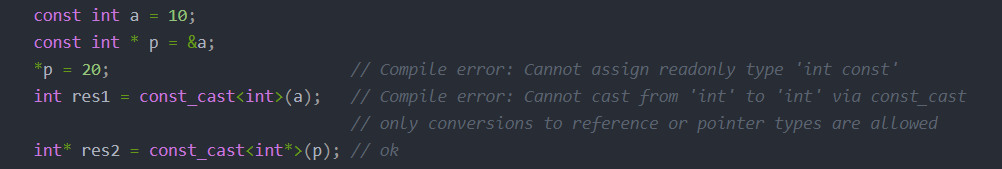
-用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

-把空指针转换成目标类型的空指针。

    -把任何类型的表达式转换成void类型。

（2）const\_cast （const\_cast<type>(expression)）

该运算符用来修改 expression 的 const 或 volatile 属性。这里需要注意：expression 和 type 的类型一样的。const\_cast<>里边的内容必须是引用或者指针



（3）reinterpret\_cast (reinterpret\_cast<type>(expression))

该运算符可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针。

 type-id 必须是一个指针、引用、算术类型、函数针或者成员指针。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。

  我们映射到的类型仅仅是为了故弄玄虚和其他目的，这是所有映射中最危险的。(这句话是C++编程思想中的原话)。因此, 你需要谨慎使用 reinterpret\_cast。

（4）dynamic\_cast

dynamic\_cast 主要用于类层次间的上行转换或下行转换。在进行上行转换时，dynamic\_cast 和 static\_cast 的效果是一样的，但在下行转换时，dynamic\_cast 具有类型检查的功能，比 static\_cast 更安全。父类指针指向子类，是安全的，而子类指向父类是不安全的，可以通过dynamic\_cast进行强制转换。

其他三种都是编译时完成的，dynamic\_cast 是运行时处理的，运行时要进行类型检查。使用 dynamic\_cast 进行转换的，**基类中一定要有虚函数**，否则编译不通过（类中存在虚函数，就说明它有想要让基类指针或引用指向派生类对象的情况，此时转换才有意义）。

**4、程序的内存模型**

（1）代码区：存放函数体的二进制代码，由操作系统进行管理的，代码区是共享的、只读的；（程序运行前）

（2）全局区（静态区）（static）：存放全局变量和静态变量以及常量（常量区、字符串常量、const修饰的全局常量和其他常量），该区域的数据在程序运行结束后由操作系统释放；（程序运行前）

（3）栈区（stack）：由编译器自动分配释放, 存放函数的参数值,局部变量等；（程序运行后）

（4）堆区（heap）：由程序员分配和释放,若程序员不释放,程序结束时由操作系统回收；（程序运行后）

BSS段：BSS段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。

数据段：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读, 某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

栈(stack)：栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

**5、堆与栈的区别**

（1）管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生 memory leak ；

（2）空间大小：一般来讲在 32 位系统下，堆内存可以达到 4G 的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在 VC6 下面，默认的栈空间大小是 1M 。当然，这个值可以修改；

（3）碎片问题：对于堆来讲，频繁的 new/delete 势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构；

（4）生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长；

（5）分配方式：堆都是动态分配的 ，没有静态分配的堆。栈的分配是由编译器完成的；

（6）分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++ 函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构 / 操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多；

**6、new和malloc的区别**

（1）属性：new和delete是C++关键字，需要编译器支持；malloc和free是库函数，需要头文件支持；

（2）参数：使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算；而malloc则需要显示地指出所需内存地尺寸；

（3）返回类型：new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符；而malloc内存分配成功则返回void\*,需要通过强转类型转换将void\*指针转换成我们需要的类型；new申请内存失败会抛出异常，malloc申请失败会返回null；

（4）自定义类型：new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现），然后调用类型的构造函数，初始化成员变量，最后返回自定义类型的指针，delete先调用析构函数，然后调用operator delete函数释放内存（通常底层用free实现）； malloc/free是库函数，只能动态地申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作；

（5）重载：C++允许重载new/delete操作符，特别的，布局new的就不需要为对象分配内存，而是指定了一个地址作为内存起始区域，new在这段内存上为对象调用构造函数完成初始化工作，并返回此地址。而malloc不允许重载。

（6）内存区域：new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局\*\*new就可以不位于堆中\*\*。

（7）malloc可以通过realloc重新分配内存，new不可以

（8）内存泄漏：对于new和malloc都能检测出内存失败，而new可以指明哪个文件的哪一行，malloc不可以；

**7、malloc的底层实现**

malloc函数的实质体现在，它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。调用malloc函数时，它沿连接表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二（一块的大小与用户请求的大小相等，另一块的大小就是剩下的字节）。接下来，将分配给用户的那块内存传给用户，并将剩下的那块（如果有的话）返回到连接表上。调用free函数时，它将用户释放的内存块连接到空闲链上。到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段，那么空闲链上可能没有可以满足用户要求的片段了。于是，malloc函数请求延时，并开始在空闲链上翻箱倒柜地检查各内存片段，对它们进行整理，将相邻的小空闲块合并成较大的内存块。

**8、C++/C中指针和引用的区别**

**引用**：C++是C语言的继承，它可进行过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。引用就是C++对C语言的重要扩充。引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。引用的声明方法：类型标识符&引用名=目标变量名；引用引入了对象的一个同义词。定义引用的表示方法与定义[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304" \t "_blank)相似，只是用&代替了\*。

**指针**：指针利用地址，它的值直接指向存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。因此，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的内存单元。

（1）指针有自己的一块空间，而引用只是一个别名；

（2）使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；

（3）指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象 的引用；

（4）作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；

（5）可以有const指针，但是没有const引用；

（6）指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；

（7）指针可以有多级指针（\*\*p），而引用至于一级；

（8）指针和引用使用++运算符的意义不一样；

（9）如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

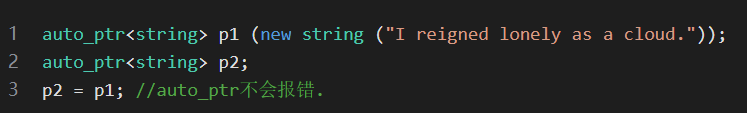
**9、智能指针**

C++里面的四个智能指针: auto\_ptr, shared\_ptr, weak\_ptr, unique\_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。（参考[详解C++11智能指针 - WindSun - 博客园 (cnblogs.com)](https://www.cnblogs.com/WindSun/p/11444429.html)

**为什么要使用智能指针：**

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

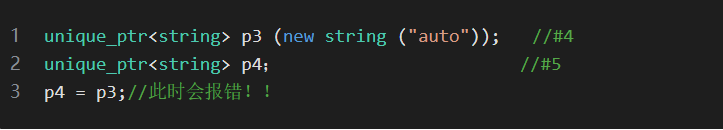
（1）auto\_ptr（C++98，C++11已放弃）



此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto\_ptr的缺点是：存在潜在的内存崩溃问题！

（2）unique\_ptr

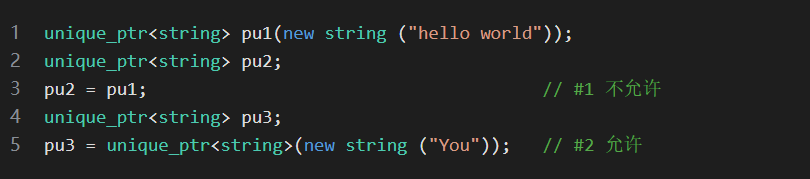
（替换auto\_ptr）unique\_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。



采用所有权模式，还是上面那个例子

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。尝试复制p3时会编译期出错，而auto\_ptr能通过编译期从而在运行期埋下出错的隐患。因此，unique\_ptr比auto\_ptr更安全。

另外unique\_ptr还有更聪明的地方：当程序试图将一个 unique\_ptr 赋值给另一个时，如果源 unique\_ptr 是个临时右值，编译器允许这么做；如果源 unique\_ptr 将存在一段时间，编译器将禁止这么做，比如：

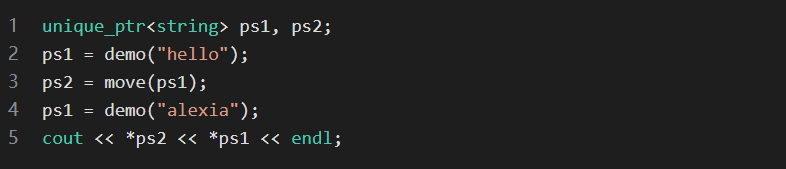


其中#1留下悬挂的unique\_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique\_ptr，因为它调用 unique\_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique\_ptr 优于允许两种赋值的auto\_ptr 。

**注：**如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique\_ptr赋给另一个。尽管转移所有权后 还是有可能出现原有指针调用（调用就崩溃）的情况。但是这个语法能强调你是在转移所有权，让你清晰的知道自己在做什么，从而不乱调用原有指针。

（**额外：**boost库的boost::scoped\_ptr也是一个独占性智能指针，但是它不允许转移所有权，从始而终都只对一个资源负责，它更安全谨慎，但是应用的范围也更狭窄。）

例如：



|  |  |
| --- | --- |
| **成员函数名** | **功 能** |
| operator\*() | 获取当前 unique\_ptr 指针指向的数据。 |
| operator->() | 重载 -> 号，当智能指针指向的数据类型为自定义的结构体时，通过 -> 运算符可以获取其内部的指定成员。 |
| operator =() | 重载了 = 赋值号，从而可以将 nullptr 或者一个右值 unique\_ptr 指针直接赋值给当前同类型的 unique\_ptr 指针。 |
| operator []() | 重载了 [] 运算符，当 unique\_ptr 指针指向一个数组时，可以直接通过 [] 获取指定下标位置处的数据。 |
| get() | 获取当前 unique\_ptr 指针内部包含的普通指针。 |
| get\_deleter() | 获取当前 unique\_ptr 指针释放堆内存空间所用的规则。 |
| operator bool() | unique\_ptr 指针可直接作为 if 语句的判断条件，以判断该指针是否为空，如果为空，则为 false；反之为 true。 |
| release() | 释放当前 unique\_ptr 指针对所指堆内存的所有权，但该存储空间并不会被销毁。 |
| reset(p) | 其中 p 表示一个普通指针，如果 p 为 nullptr，则当前 unique\_ptr 也变成空指针；反之，则该函数会释放当前 unique\_ptr 指针指向的堆内存（如果有），然后获取 p 所指堆内存的所有权（p 为 nullptr）。 |
| swap(x) | 交换当前 unique\_ptr 指针和同类型的 x 指针。 |

（3）shared\_ptr

shared\_ptr实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use\_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto\_ptr, unique\_ptr,weak\_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

对shared\_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make\_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针。

shared\_ptr 是为了解决 auto\_ptr 在对象所有权上的局限性(auto\_ptr 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

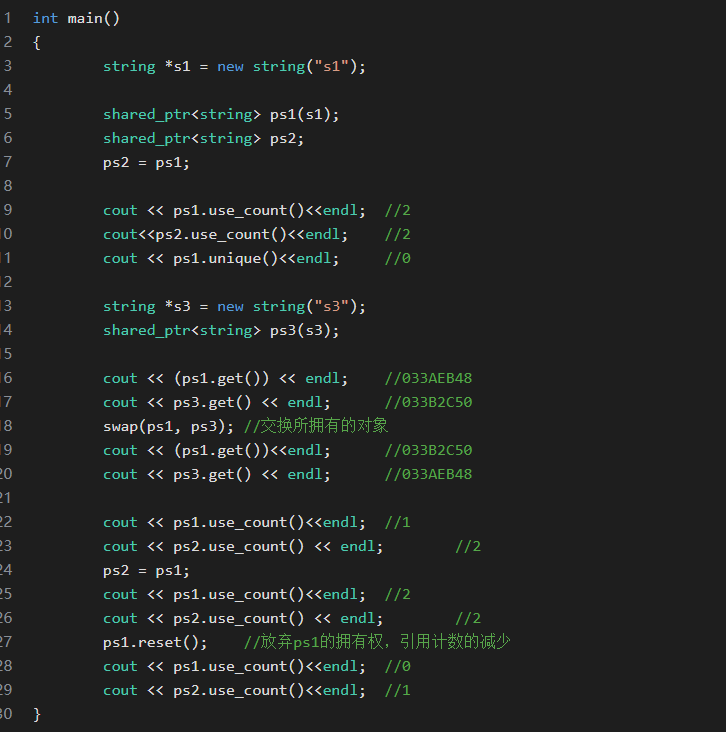
use\_count 返回引用计数的个数

unique 返回是否是独占所有权( use\_count 为 1)

swap 交换两个 shared\_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更, 会引起原有对象的引用计数的减少

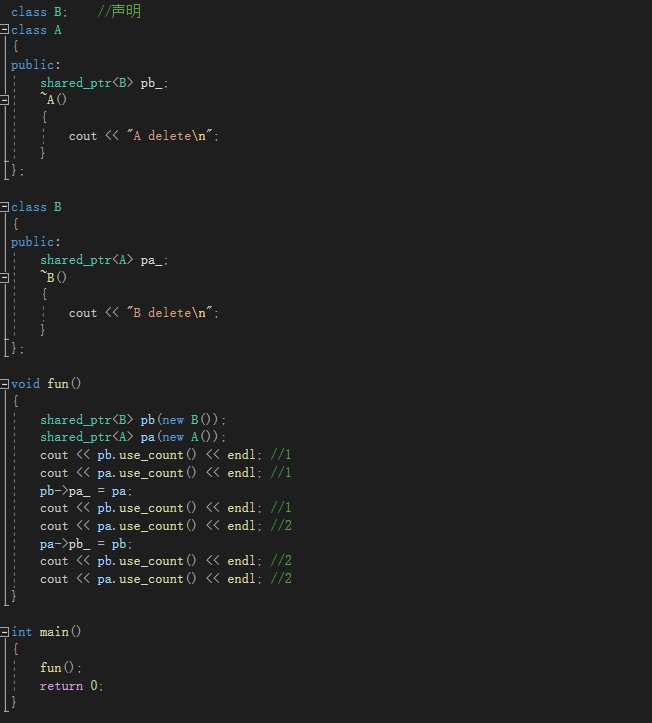
get 返回内部对象(指针), 由于已经重载了()方法, 因此和直接使用对象是一样的.如



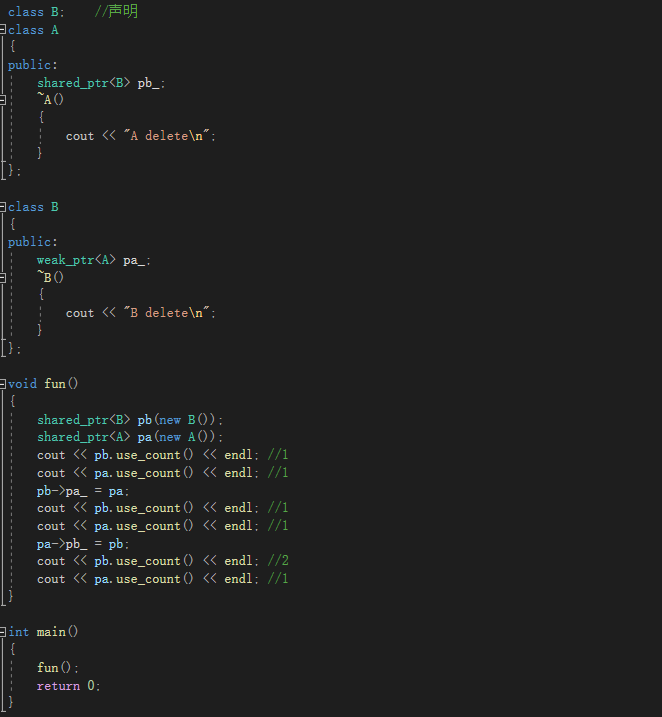
（4）weaked\_ptr

share\_ptr虽然已经很好用了，但是有一点share\_ptr智能指针还是有内存泄露的情况，当两个对象相互使用一个shared\_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。

weak\_ptr 是一种不控制对象生命周期的智能指针, 它指向一个shared\_ptr管理的对象. 进行该对象的内存管理的是那个强引用的shared\_ptr， weak\_ptr只是提供了对管理对象的一个访问手段。weak\_ptr 设计的目的是为配合 shared\_ptr 而引入的一种智能指针来协助 shared\_ptr 工作, 它只可以从一个 shared\_ptr 或另一个 weak\_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。weak\_ptr是用来解决shared\_ptr相互引用时的死锁问题,如果说两个shared\_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared\_ptr之间可以相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr。

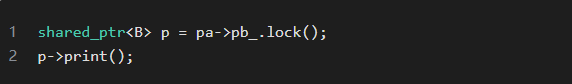


可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减1，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A、B的析构函数没有被调用）运行结果没有输出析构函数的内容，造成内存泄露。如果把其中一个改为weak\_ptr就可以了，我们把类A里面的shared\_ptr pb\_，改为weak\_ptr pb\_ ，运行结果如下：



这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减1，同时pa析构时使A的计数减1，那么A的计数为0，A得到释放。

**注意**：我们不能通过weak\_ptr直接访问对象的方法，比如B对象中有一个方法print()，我们不能这样访问，pa->pb\_->print()，因为pb\_是一个weak\_ptr，应该先把它转化为shared\_ptr，如：



weak\_ptr 没有重载\*和->但可以使用 lock 获得一个可用的 shared\_ptr 对象. 注意, weak\_ptr 在使用前需要检查合法性.

expired 用于检测所管理的对象是否已经释放, 如果已经释放, 返回 true; 否则返回 false.

lock 用于获取所管理的对象的强引用(shared\_ptr). 如果 expired 为 true, 返回一个空的 shared\_ptr; 否则返回一个 shared\_ptr, 其内部对象指向与 weak\_ptr 相同.

use\_count 返回与 shared\_ptr 共享的对象的引用计数.

reset 将 weak\_ptr 置空.

weak\_ptr 支持拷贝或赋值, 但不会影响对应的 shared\_ptr 内部对象的计数.

**10、数组与指针的区别**

|  |  |
| --- | --- |
| 指针 | 数组 |
| 保存数据的地址 | 保存数据 |
| 间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据 | 直接访问数据， |
| 通常用于动态的数据结构 | 通常用于固定数目且数据类型相同的元素 |
| 通过Malloc分配内存，free释放内存 | 隐式的分配和删除 |
| 通常指向匿名数据，操作匿名函数 | 自身即为数据名 |
| 求sizeof是4字节 | 求sizeof是数组个数X数组成员所占字节数 |

**11、野指针与空指针**

- 野指针：指向一个已删除的对象或者申请访问首先内存区域的指针；

- 空指针：空指针不指向任何实际的对象或者函数。反过来说，任何对象或者函数的地址都不可能是空指针。

**12、请你说一下strcpy和strlen**

- strcpy是字符串拷贝函数，原型：char \*strcpy(char\* dest, const char\* src)

从src逐字节拷贝到dest，直到遇到'\0'结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞,安全版本是strncpy函数。  
- strlen函数是计算字符串长度的函数，返回从开始到'\0'之间的字符个数。

**13、C++中析构函数的作用**

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。

析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。

如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。

类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

**14、C++多态**

多态性(polymorphism)可以简单地概括为“一个接口，多种方法”，它是面向对象编程领域的核心概念。

多态性指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。

1. 编译时多态性（静态多态）：通过重载函数实现：先期联编 early binding
2. 运行时多态性（动态多态）：通过虚函数实现 ：滞后联编 late binding

C++运行时多态性是通过虚函数来实现的，虚函数允许子类重新定义成员函数，而子类重新定义父类的做法称为覆盖(Override)，或者称为重写。

静态多态与动态多态的实质区别就是函数地址是早绑定还是晚绑定。如果函数的调用，在编译器编译期间就可以确定函数的调用地址，并生产代码，是静态的，就是说地址是早绑定的。而如果函数调用的地址不能在编译器期间确定，需要在运行时才确定，这就属于晚绑定。

多态的目的：封装可以使得代码模块化，继承可以扩展已存在的代码，他们的目的都是为了代码重用。而多态的目的则是为了“接口重用”。也即，不论传递过来的究竟是类的哪个对象，函数都能够通过同一个接口调用到适应各自对象的实现方法。

多态最常见的用法就是声明基类类型的指针，利用该指针指向任意一个子类对象，调用相应的虚函数，可以根据指向的子类的不同而实现不同的方法。如果没有使用虚函数的话，即没有利用C++多态性，则利用基类指针调用相应的函数的时候，将总被限制在基类函数本身，而无法调用到子类中被重写过的函数。因为没有多态性，函数调用的地址将是固定的，因此将始终调用到同一个函数，这就无法实现“一个接口，多种方法”的目的了。

**需要注意：**

--只有类的成员函数才能声明为虚函数，虚函数仅适用于有继承关系的类对象。普通函数不能声明为虚函数。

--静态成员函数不能是虚函数，因为静态成员函数不受限于某个对象。

--内联函数（inline）不能是虚函数，因为内联函数不能在运行中动态确定位置。

--构造函数不能是虚函数。

--析构函数可以是虚函数，而且建议声明为虚函数。

**15、重载与重写**

- 重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中

- 重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写

**16、虚函数实现机制**

虚函数：虚函数由于必须是在类中声明的函数，因此又称为虚方法。所有以virtual修饰符开始的成员函数都成为虚方法。此时注意是virtual修饰的成员函数不是virtual修饰的成员函数名。

基类的虚函数表的创建：首先在基类声明中找到所有的虚函数，按照其声明顺序，编码0,1,2,3,4……，然后按照此声明顺序为基类创建一个虚函数表，其内容就是指向这些虚函数的函数指针，按照虚函数声明的顺序将这些虚函数的地址填入虚函数表中。例如若show放在虚函数声明的第二位，则在虚函数表中也放在第二位。

对于子类的虚函数表：首先将基类的虚函数表复制到该子类的虚函数表中。若子类重写了基类的虚函数show，则将子类的虚函数表中存放show的函数地址(未重写前存放的是子类的show虚函数的函数地址)更新为重写后函数的函数指针。若子类增加了一些虚函数的声明，则将这些虚函数的地址加到该类虚函数表的后面。

总结：当调用pBase->show();时，执行的步骤：

判断Base类中show是否为虚函数。若不是虚函数则找到pBase所指向的对象所属类Base。执行Base::show()。若是虚函数则访问pBase所指对象的虚函数表指针得到pBase所指对象所在类的虚函数表。查找Base中show()在声明时的位序为x，到步骤3得到的虚函数表中找到位序x，从而得到要执行的show的函数地址。根据函数地址和Base中声明的show的函数类型(形参和返回值)访问地址所指向的函数。

**17、为什么析构函数必须是虚函数？为什么C++默认的析构函数不是虚函数？**

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，否则则只会释放基类的空间，而不会释放子类的部分空间，造成内存泄漏。

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数。

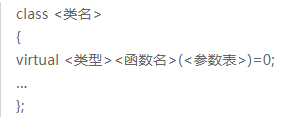
**18、友元能否是虚函数？**

友元不能是虚函数，因为友元不是类成员，只有成员函数才能是虚函数。

**19、纯虚函数**

纯虚函数也可以叫抽象函数。一般来说它只有函数名、参数和返回值类型，不需要函数体。

纯虚函数是一种特殊的虚函数，它的一般格式如下(C++格式)：



纯虚函数的意义，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的默认实现。所以类纯虚函数的声明就是在告诉子类的设计者，"你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它" 。当想在基类中抽象出一个方法，且该基类只做能被继承，而不能被实例化；这个方法必须在派生类(derived class)中被实现；

**20、静态函数与虚函数的区别**

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销