

**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。

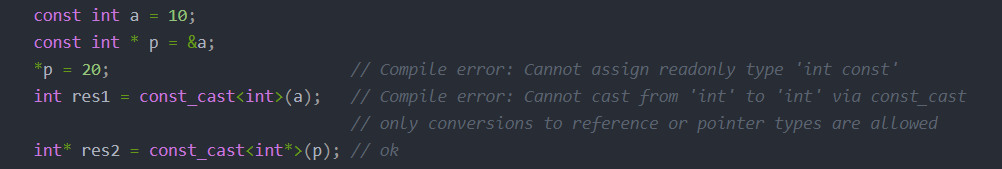
-用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

-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。由于栈的先进后出特点，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)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），而引用只有一级；

（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。

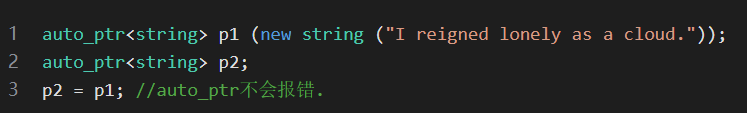
**9、智能指针**

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。

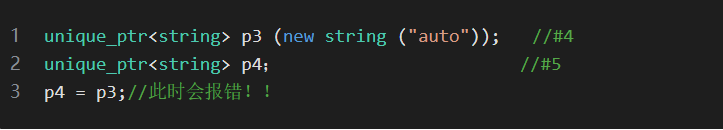
（1）auto\_ptr（C++98，C++11已放弃）



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

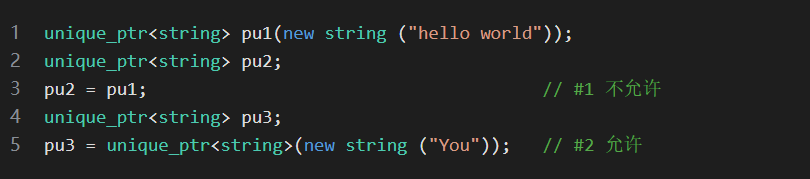
（替换auto\_ptr）unique\_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。



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：

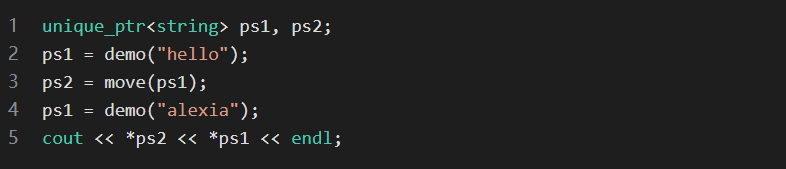


其中#1留下悬挂的unique\_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique\_ptr，因为它调用 unique\_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique\_ptr 优于允许两种赋值的auto\_ptr 。

**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：

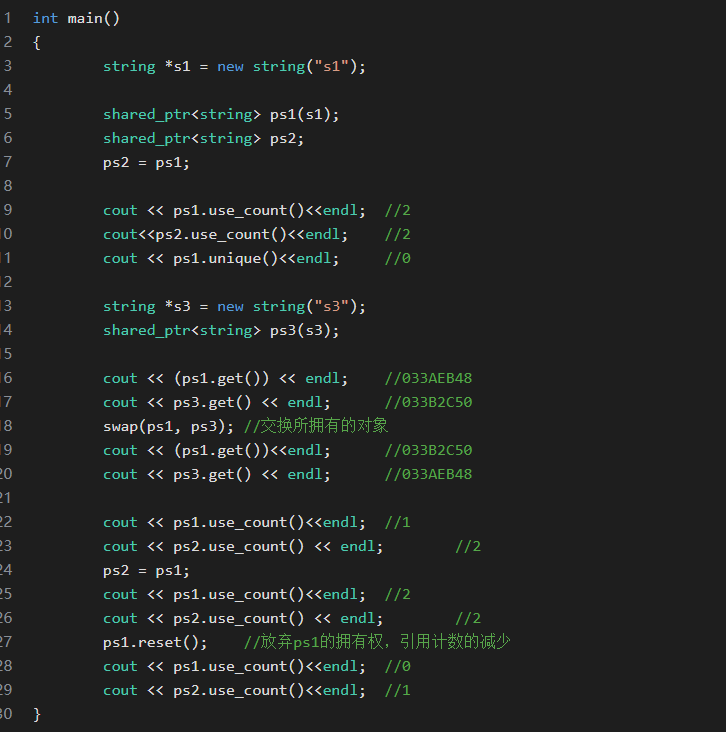
use\_count 返回引用计数的个数

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

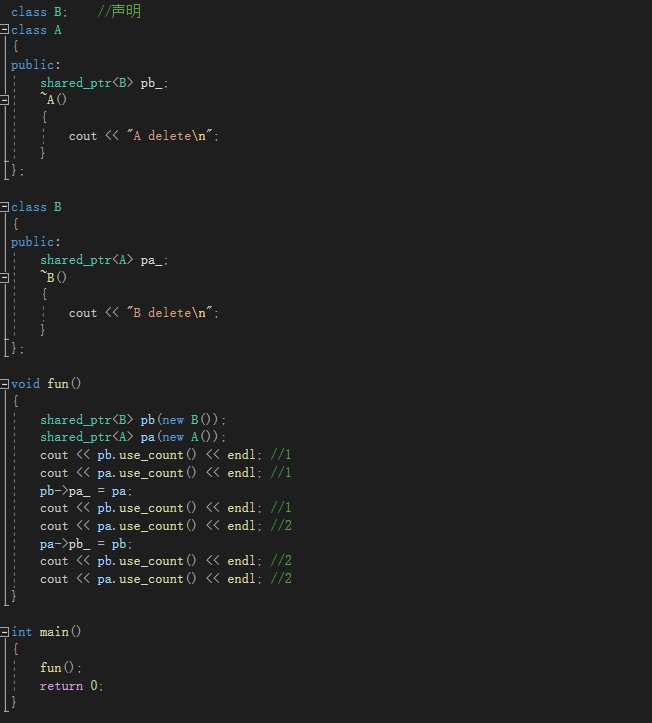
get 返回内部对象(指针), 由于已经重载了()方法, 因此和直接使用对象是一样的.如



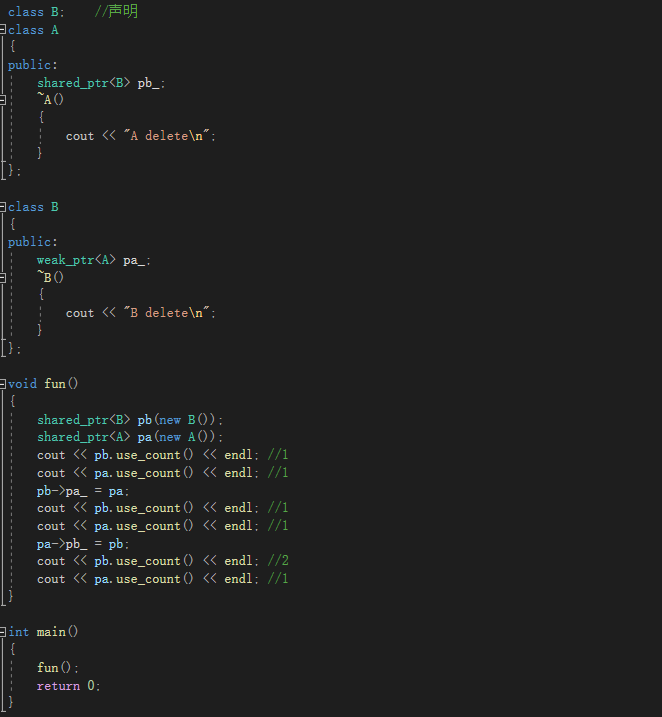
（4）weaked\_ptr

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。

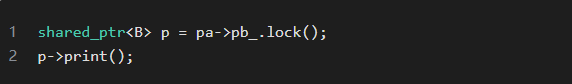


可以看到fun函数中pa，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减1，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A、B的析构函数没有被调用）运行结果没有输出析构函数的内容，造成内存泄露。如果把其中一个改为weak\_ptr就可以了，我们把类A里面的shared\_ptr pb\_，改为weak\_ptr pb\_ ，运行结果如下：



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**

1）野指针：指向一个已删除的对象或者未申请访问受限内存区域的指针；

2）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**

1）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。

2）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。

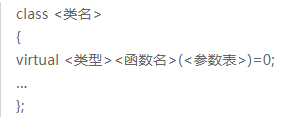
**18、友元能否是虚函数？**

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。

**21、C++显性转换和隐性转换**

- 隐性转换：低精度变量赋值给高精度变量（如int转为float）, 其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象

- 显性转换：其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。比如int转char；C++类对象的显式转换**：**当类构造函数只有一个参数或除了第一个参数外其余参数都有默认值时，则此类有隐含的类型转换操作符(隐式转换)，但有时隐式转换并不是我们想要的，可在构造函数前加上关键字explicit，来指定显式调用。

**22、C语言函数参数压栈顺序**是从由右到左。栈是向下生长的，即从高地址向低地址的方向分配内存呢。

**23、C++中类成员的访问权限**

C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员。protected受保护的，只有派生类和内部可以访问。

**24、C++中struct和class的区别**

在C++中，可以用struct和class定义类，都可以继承。区别在于：structural的默认继承权限和默认访问权限是public，而class的默认继承权限和默认访问权限是private。

另外，class还可以定义模板类形参，比如template <class T, int i>。

**25、C++类内可以引用数据成员**

通过成员函数初始化列表初始化。

**26、右值引用**

右值引用是C++11中引入的新特性 , 它实现了转移语义和精确传递。它的主要目的有两个方面：

1）消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。

2）能够更简洁明确地定义泛型函数。

左值和右值的概念：

左值：能对表达式取地址、或具名对象/变量。一般指表达式结束后依然存在的持久对象。

右值：不能对表达式取地址，或匿名对象。一般指表达式结束就不再存在的临时对象。

右值引用和左值引用的区别：

1）左值可以寻址，而右值不可以。

2）左值可以被赋值，右值不可以被赋值，可以用来给左值赋值。

3）左值可变，右值不可变（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）。

**27、C++源文件从文本到可执行文件经历的过程**

对于C++源文件，从文本到可执行文件一般需要四个过程：

预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。

编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件

汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

**28、include头文件的顺序以及双引号””和尖括号<>的区别**

Include头文件的顺序：对于include的头文件来说，如果在文件a.h中声明一个在文件b.h中定义的变量，而不引用b.h。那么要在a.c文件中引用b.h文件，并且要先引用b.h，后引用a.h,否则汇报变量类型未声明错误。

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。

1）对于使用双引号包含的头文件，查找头文件路径的顺序为：

当前头文件目录

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS\_INCLUDE\_PATH/C\_INCLUDE\_PATH指定的头文件路径

2）对于使用尖括号包含的头文件，查找头文件的路径顺序为：

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS\_INCLUDE\_PATH/C\_INCLUDE\_PATH指定的头文件路径

**29、什么时候发生段错误**

1）段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况：

2）使用野指针

3）试图修改字符串常量的内容

**21、类模板与继承**

1）当子类继承的父类是一个类模板时，子类在声明的时候，需要指出父类中T的类型；

2）如果不指定，编译器无法给子类分配内存；

3）如果想灵活指出父类中T的类型，子类也需要编程类模板；

**22、深拷贝和浅拷贝**

1）在未定义显示拷贝构造函数的情况下，系统会调用默认的拷贝函数——即浅拷贝，它能够完成成员的一一复制。当数据成员中没有指针时，浅拷贝是可行的；但当数据成员中有指针时，如果采用简单的浅拷贝，则两类中的两个指针将指向同一个地址，当对象快结束时，会调用两次析构函数，而导致指针悬挂现象，所以，此时，必须采用深拷贝。

2）深拷贝与浅拷贝的区别就在于深拷贝会在堆内存中另外申请空间来储存数据，从而也就解决了指针悬挂的问题。简而言之，当数据成员中有指针时，必须要用深拷贝

**二、容器和算法**

**1、map和set的区别，分别又是如何实现的？**

map和set都是C++的关联容器，其底层实现都是红黑树（RB-Tree）。由于map和set所开放的各种操作接口，RB-tree也都提供了，所以几乎所有的map和set的操作行为，都只是转调RB-tree的操作行为。

map和set区别在于：

1）map中的元素是key-value（关键字—值）对：关键字起到索引的作用，值则表示与索引相关联的数据；Set与之相对就是关键字的简单集合，set中每个元素只包含一个关键字。

2）set的迭代器是const的，不允许修改元素的值；map允许修改value，但不允许修改key。其原因是因为map和set是根据关键字排序来保证其有序性的，如果允许修改key的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了map和set的结构，导致iterator失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以STL中将set的迭代器设置成const，不允许修改迭代器的值；而map的迭代器则不允许修改key值，允许修改value值。

3）map支持下标操作，set不支持下标操作。map可以用key做下标，map的下标运算符[ ]将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和mapped\_type类型默认值的元素至map中，因此下标运算符[ ]在map应用中需要慎用，const\_map不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped\_type类型没有默认值也不应该使用。如果find能解决需要，尽可能用find。

**2、vector和list的区别**

1）vector

连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：

vector增加（插入）新元素时，如果未超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会失效。

性能：

访问：O(1)

插入：在最后插入（空间够）：很快

在最后插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

在中间插入（空间够）：内存拷贝

在中间插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

删除：在最后删除：很快

在中间删除：内存拷贝

适用场景：经常随机访问，且不经常对非尾节点进行插入删除。

2) List

动态链表，在堆上分配空间，每插入一个元数都会分配空间，每删除一个元素都会释放空间。

底层：双向链表

性能：

访问：随机访问性能很差，只能快速访问头尾节点。

插入：很快，一般是常数开销

删除：很快，一般是常数开销

适用场景：经常插入删除大量数据

3）区别

vector底层实现是数组；list是双向链表。

vector支持随机访问，list不支持。

vector是顺序内存，list不是。

vector在中间节点进行插入删除会导致内存拷贝，list不会。

vector一次性分配好内存，不够时才进行2倍扩容；list每次插入新节点都会进行内存申请。

vector随机访问性能好，插入删除性能差；list随机访问性能差，插入删除性能好。

4）应用

vector拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随即访问，而不在乎插入和删除的效率，使用vector。

list拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，则应使用list。

**3、STL组成**

STL主要由：以下几部分组成：

容器containers、迭代器Iterator adapters、仿函数functors、算法algorithms、分配器allocator、配接器container adapters

它们之间的关系：分配器给容器分配存储空间，算法通过迭代器获取容器中的内容，仿函数可以协助算法完成各种操作，配接器用来套接适配仿函数；

**4、STL中resize和reserve的区别**

resize()：改变当前容器内含有元素的数量(size())，eg: vector<int>v; v.resize(len);v的size变为len,如果原来v的size小于len，那么容器新增（len-size）个元素，元素的值为默认为0.当v.push\_back(3);之后，则是3是放在了v的末尾，即下标为len，此时容器是size为len+1；

reserve()：改变当前容器的最大容量（capacity）,它不会生成元素，只是确定这个容器允许放入多少对象，如果reserve(len)的值大于当前的capacity()，那么会重新分配一块能存len个对象的空间，然后把之前v.size()个对象通过copy construtor复制过来，销毁之前的内存；

**5、STL中allocator**

STL的分配器用于封装STL容器在内存管理上的底层细节。在C++中，其内存配置和释放如下：

new运算分两个阶段：(1)调用:operator new配置内存;(2)调用对象构造函数构造对象内容

delete运算分两个阶段：(1)调用对象析构函数；(2)调用:operator delete释放内存

为了精密分工，STL allocator将两个阶段操作区分开来：内存配置由alloc::allocate()负责，内存释放由alloc::deallocate()负责；对象构造由::construct()负责，对象析构由::destroy()负责。

同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL采用了两级配置器，当分配的空间大小超过128B时，会使用第一级空间配置器；当分配的空间大小小于128B时，将使用第二级空间配置器。第一级空间配置器直接使用malloc()、realloc()、free()函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存。

**6、STL迭代器iterator**

迭代器(iterator)是一种抽象的设计理念，通过迭代器可以在不了解容器内部原理的情况下遍历容器。除此之外，STL中迭代器一个最重要的作用就是作为容器(vector,list等)与STL算法的粘结剂，只要容器提供迭代器的接口，同一套算法代码可以利用在完全不同的容器中，这是抽象思想的经典应用。

不同的容器其对应的迭代器也是不同的

2）迭代器与指针的区别

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，->、\*、++、--等。迭代器封装了指针，是一个“可遍历STL（ Standard Template Library）容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。

迭代器返回的是对象引用而不是对象的值，所以cout只能输出迭代器使用\*取值后的值而不能直接输出其自身。

3）迭代器产生原因

Iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

**7、vector**

1）push\_back：将元素插入末尾时，检测是否有备用空间，若有则直接插入，若没有，重新分配一个原有2倍的内存空间，并复制数据，并释放原有的空间；

2）erase、insert：造成后面的迭代器全部失效；

3）vector是连续线性空间，单向开口；迭代器是随机访问；

**8、list**

1）环状双向链表，提供的迭代器类型是Bidirectional Iterators

2）插入insert、接合splice不会造成原有的迭代器失效，删除erase只会使被删元素的迭代器失效

**9、deque**

1）双向开口的连续线性空间，可以在开头和结尾分别做元素的插入和删除操作

2）与vector最大的差异：在于deque允许常数时间内对起头端进行元素的插入或移除操作，以及deque没有容量capacity概念，因为其是动态地以分段连续空间组合而成。

3）deque中进行排序操作，为了最高效率，是将deque复制到一个vector容器，将vector排序后再复制回deque；

4）deque数据结构：除了维护一个指向map的指针外，也要维护start、finish的两个迭代器，分别指向第一个缓存区的第一个元素和最后一个缓存区的最后一个元素

**10、stack**

1）先进后出（FILO）的数据结构，允许新增、移除、获取最顶端元素；不允许有遍历行为；没有迭代器

2）以deque为底部结构并封闭其头端出口，便形成stack。（具有”修改某物接口，形成另一种风貌“性质者，成为adapter配接器，stack是配接器）；除deque外还可以使用list；

**11、queue**

1）先进先出（FIFO）的数据结构，允许最顶端取出、最底端加入、获取最顶端元素；不允许有遍历行为；没有迭代器

**12、set**

1）所有元素会根据元素的键值自动排序；其元素的键值就是实值；元素值不可以更改；

2）底层实现是红黑树（RB-Tree）;

3）面对关联式容器，应该使用其所提供的find函数来搜寻元素，会比使用STL算法find（）更有效率，因为STL算法的find（）只是循序搜寻；

**13、map**

1）所有元素都会根据元素的键值对自动被排序；

2）map所有元素都是pair；map不允许两个元素拥有相同的键值；map的键值不能通过迭代器进行修改，因为改变key会改变map组织，但是可以修改实值value；

3）底层实现是红黑树；

**14、multiset**

1）与set用法完全相同；

2）唯一差别在于它允许键值重复

**15、hashtable**

1）字典结构；

2）拉链法解决哈希冲突

**16、算法的泛化过程，int \*find(int \*arrayHead, int arraySize, int value)**

查找到则返回找到他的第一个的指针，查找不到则返回最后一个元素的下一位置（地址），不返回null的原因是为了带来泛化效果。

**17、仿函数**

仿函数实际上是一个“行为类似函数“的对象。仿函数主要应用在算法的实现里，用来实现一个算法的多种功能，比如sort函数，通过仿函数机制使得用户可以自定义排序规则。。仿函数主要用于STL中的算法中，虽然函数指针虽然也可以作为算法的参数，但是函数指针不能满足STL对抽象性的要求，也不能满足软件积木的要求–函数指针无法和STL其他组件搭配，产生更灵活变化。仿函数本质就是类重载了一个operator()，创建一个行为类似函数的对象。

**18、配接器**

应用于容器：container adapter，如stack、queue；

应用于迭代器：iterator adapter；

应用于仿函数：function adapter；

**19、STL迭代器如何删除元素的？**

这个主要考察的是迭代器失效的问题。

1）对于序列容器vector,deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；

2）对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

3）对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator，因此上面两种正确的方法都可以使用。

**20、emplace\_back和push\_back的区别？**

emplace\_back() 和 push\_back() 的区别，就在于底层实现的机制不同。在C++ STL中，push\_back()函数向容器中加入一个临时对象（右值元素）时，首先会调用构造函数生成这个对象，然后调用拷贝构造函数将这个对象放入容器中，最后释放临时对象。而emplace\_back()函数向容器中加入临时对象，临时对象原地构造，没有拷贝或移动的操作，所以效率比较高。一般用emplace\_back取代push\_back。

**21、排序算法**

1）属于稳定排序的有：直接插入排序、简单选择排序、归并排序、基数排序；

属于不稳定排序的有：希尔排序、快速排序、堆排序；

2）算法性能

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 排序方法 | 最好时间 | 平均时间 | 最坏时间 | 空间复杂度 |
| 直接插入排序 | O(n) | O(n^2) | O(n^2) | O(1) |
| 希尔排序 | O(n^1.3) | O(nlog2n) | O(n^2) | O(1) |
| 快速排序 | O(nlog2n) | O(nlog2n) | O(n^2) | O(nlgn) |
| 直接选择排序 | O(n^2) | O(n^2) | O(n^2) | O(1) |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) |
| 归并排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(n) |
| 基数排序 | O(d(n+rd)) | O(d(n+rd)) | O(d(n+rd)) | O(n+rd) |

**三、数据结构**

**1、平衡二叉树（AVL树）**

平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。一句话表述为：以树中所有结点为根的树的左右子树高度之差的绝对值不超过1。将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

**2、红黑树**

红黑树和我们以前学过的AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。AVL树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

红黑树和AVL树的区别在于它使用颜色来标识结点的高度，它所追求的是局部平衡而不是AVL树中的非常严格的平衡。学过数据结构的人应该都已经领教过AVL树的复杂，但AVL树的复杂比起红黑树来说简直是小巫见大巫，红黑树才是真正的变态级数据结构。

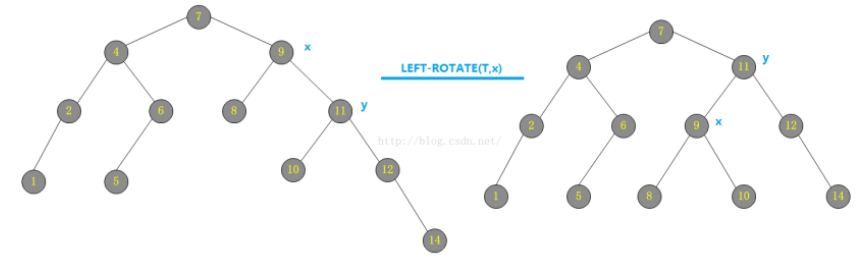
红黑树是AVL树的变种，红黑树通过一些着色法则确保没有一条路径会比其它路径长出两倍，因而达到接近平衡的目的。所谓红黑树，不仅是一个二叉搜索树，而且必须满足一下规则：

1. 每个节点不是红色就是黑色。
2. 根节点为黑色。
3. 如果节点为红色，其子节点必须为黑色。
4. 任意一个节点到到NULL（树尾端）的任何路径，所含之黑色节点数必须相同。

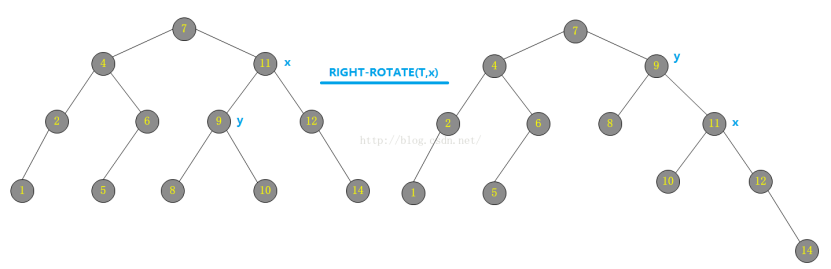
上面的这些约束保证了这个树大致上是平衡的，这也决定了红黑树的插入、删除、查询等操作是比较快速的。 根据规则4，新增节点必须为红色；根据规则3，新增节点之父节点必须为黑色。当新增节点根据二叉搜索树的规则到达其插入点时，却未能符合上述条件时，就必须调整颜色并旋转树形。

红黑树的旋转：

1）左旋：对某个结点x做左旋操作时，假设其右孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的左孩子，y的左孩子成为x的右孩子。



2）右旋：对某个结点x做右旋操作时，假设其左孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的右孩子，y的右孩子成为x的左孩子。



**3、B树和B+树**

B树的出现是为了弥合不同的存储级别之间的访问速度上的巨大差异，实现高效的 I/O。平衡二叉树的查找效率是非常高的，并可以通过降低树的深度来提高查找的效率。但是当数据量非常大，树的存储的元素数量是有限的，这样会导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下。另外数据量过大会导致内存空间不够容纳平衡二叉树所有结点的情况。B树是解决这个问题的很好的结构

M阶B树满足条件：

* 每个节点最多只有m个子节点。
* 每个非叶子节点（除了根）具有至少⌈ m/2⌉子节点。
* 如果根不是叶节点，则根至少有两个子节点。
* 具有*k*个子节点的非叶节点包含*k* -1个键。
* 所有叶子都出现在同一水平，没有任何信息（高度一致）。

B+树的特征：

* 有m个子树的中间节点包含有m个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引；
* 所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。 (而B 树的叶子节点并没有包括全部需要查找的信息)；
* 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B 树的非终节点也包含需要查找的有效信息)；

B+树相比B树的优势：

　　1）单一节点存储更多的元素，使得查询的IO次数更少；

　　2）所有查询都要查找到叶子节点，查询性能稳定；

3）所有叶子节点形成有序链表，便于范围查询。

**4、map底层为什么用红黑树实现**

AVL树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是O(logn)，且性能稳定，所以STL里面很多结构包括map底层实现都是使用的红黑树。

**5、map和unordered\_map的优点和缺点**

map优点：

对于map，其底层是基于红黑树实现的，优点如下：

1)有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作

2）map的查找、删除、增加等一系列操作时间复杂度稳定，都为logn

map缺点：

1. 查找、删除、增加等操作平均时间复杂度较慢，与n相关

unordered\_map优点

其底层是一个哈希表,查找、删除、添加的速度快，时间复杂度为常数级O(c)

unordered\_map缺点：

unordered\_map内部基于哈希表，以（key,value）对的形式存储，因此空间占用率高。 unordered\_map的查找、删除、添加的时间复杂度不稳定，平均为O(c)，取决于哈希函数。极端情况下可能为O(n)。

**6、栈溢出**

栈溢出概念：

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致栈中与其相邻的变量的值被改变。

栈溢出的原因：

1）局部数组过大。当函数内部的数组过大时，有可能导致堆栈溢出。局部变量是存储在栈中的，因此这个很好理解。解决这类问题的办法有两个，一是增大栈空间,二是改用动态分配，使用堆（heap）而不是栈（stack）。

2）递归调用层次太多。递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致堆栈溢出。

3）指针或数组越界。这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。

**7、hash表的实现以及解决哈希冲突的方法**

hash表的实现主要包括构造哈希和处理哈希冲突两个方面：

对于构造哈希来说，主要包括直接地址法、平方取中法、除留余数法等。

对于处理哈希冲突来说，最常用的处理冲突的方法有开放定址法、再哈希法、链地址法、建立公共溢出区等方法。SGL版本使用链地址法，使用一个链表保持相同散列值的元素。

虽然链地址法并不要求哈希桶长度必须为质数，但SGI STL仍然以质数来设计哈希桶长度，并且将28个质数（逐渐呈现大约两倍的关系）计算好，以备随时访问，同时提供一个函数，用来查询在这28个质数之中，“最接近某数并大于某数”的质数。

当哈希表关键字集合很大时，关键字值不同的元素可能会映象到哈希表的同一地址上，这样的现象称为哈希冲突。目前常用的解决哈希冲突的方法如下：

1）开放定址法: 当发生地址冲突时，按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。

2）再哈希法：当发生哈希冲突时使用另一个哈希函数计算地址值，直到冲突不再发生。这种方法不易产生聚集，但是增加计算时间，同时需要准备许多哈希函数。

3）链地址法：将所有哈希值相同的Key通过链表存储。key按顺序插入到链表中

4）建立公共溢出区：采用一个溢出表存储产生冲突的关键字。如果公共溢出区还产生冲突，再采用处理冲突方法处理。

**链地址法特点**

* 拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；
* 由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；

STL底层的hashMap是由数组和链表来实现的，就是上面说的拉链法。首先当插入的时候，会根据key的hash值然后计算出相应的数组下标，计算方法是index = hashcode%table.length，（这个下标就是上面提到的bucket），当这个下标上面已经存在元素的时候那么就会形成链表，将后插入的元素放到尾端，若是下标上面没有存在元素的话，那么将直接将元素放到这个位置上。   
当进行查询的时候，同样会根据key的hash值先计算相应的下标，然后到相应的位置上进行查找，若是这个下标上面有很多元素的话，那么将在这个链表上一直查找直到找到对应的元素。