[**new和malloc的区别**](http://blog.csdn.net/nie19940803/article/details/76358673)



指针和引用区别

引用必须初始化，指针可以不用

指针传参数时是值传递，引用传参时就是自己

指针是一个实体他在栈中有自己使用的空间，但是引用没有

#define和const区别

define在预处理阶段起作用，而const是在编译时;const有类型检查而define没有，只是进行简单的字符串替换；const定义常数是变量带类型，define只是个常数不带类型

**C++中有malloc/free，为什么还有new/delete？**

对于类类型的对象而言，用malloc/free无法满足要求的。对象在创建的时候要自动执行构造函数，消亡之前要调用析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制之内，不能把执行构造函数和析构函数的任务强加给它，因此，C++还需要new/delete。

**C++new和delete实现原理**

[**https://blog.csdn.net/passion\_wu128/article/details/38966581**](https://blog.csdn.net/passion_wu128/article/details/38966581)

**空指针和悬垂指针的区别？**

空指针是指被赋值为NULL的指针；delete指向动态分配对象的指针将会产生悬垂指针。（1） 空指针可以被多次delete，而悬垂指针再次删除时程序会变得非常不稳定；

（2） 使用空指针和悬垂指针都是非法的，而且有可能造成程序崩溃，如果指针是空指针，尽管同样是崩溃，但和悬垂指针相比是一种可预料的崩溃。

**面向对象技术的基本概念是什么，三个基本特征是什么？**

基本概念：类、对象、继承； 基本特征：封装、继承、多态。

**哪一种成员变量可以在一个类的实例之间共享？**

static静态成员变量

**继承层次中，为什么基类析构函数是虚函数？**

编译器总是根据类型来调用类成员函数。但是一个派生类的指针可以安全地转化为一个基类的指针。这样删除一个基类的指针的时候，C++不管这个指针指向一个基类对象还是一个派生类的对象，调用的都是基类的析构函数而不是派生类的。如果你依赖于派生类的析构函数的代码来释放资源，而没有重载析构函数，那么会有资源泄漏。

**为什么构造函数不能为虚函数？构造函数可以为内联函数**

虚函数采用一种虚调用的方法。虚调用是一种可以在只有部分信息的情况下工作的机制。如果创建一个对象，则需要知道对象的准确类型，因此构造函数不能为虚函数。

**重载和覆盖有什么区别？（书p509）**

虚函数是基类希望派生类重新定义的函数，派生类重新定义基类虚函数的做法叫做覆盖；

重载就在允许在相同作用域中存在多个同名的函数，这些函数的参数表不同。重载的概念不属于面向对象编程，编译器根据函数不同的形参表对同名函数的名称做修饰，然后这些同名函数就成了不同的函数。

重载的确定是在编译时确定，是静态的；虚函数则是在运行时动态确定。

如果重新定义派生类的函数将不只是使用相同的函数参数列表覆盖基类声明，无论列表是否相同，该操作将隐藏同名基类方法。如果基类声明被重载了，则应在派生类中重新定义所有基类版本。

**公有继承时基类受保护的成员，可以通过派生类对象访问但不能修改。**

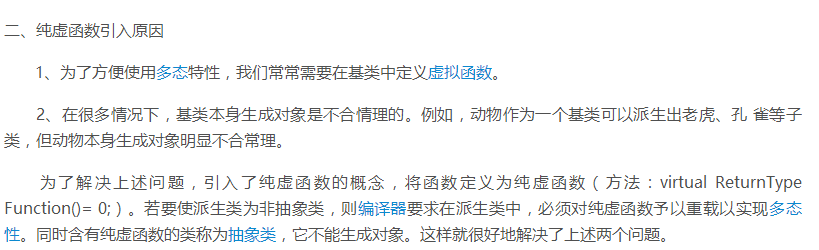
**Is-a public; has-a private/protected/包含**

**C++如何阻止一个类被实例化？一般在什么时候将构造函数声明为private？**

将类定义为抽象基类或者将构造函数声明为private；不允许类外部创建类对象，只能在类内部创建对象

**抽象基类**

抽象类只能作为基类来使用.抽象基类必须至少使用一个纯虚函数接口。(纯虚函数可以在抽象类中定义)抽象基类不能创建对象实例.只能当基类被其它类继承。原型中使用=0指出类是一个抽象基类，在类中可以不定义该函数。抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类没有重新定义纯虚函数，而派生类只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体类了.基类的纯虚函数必须有“=0”，但不一定没有函数的实现，只是不能直接内嵌在类中.抽象基类的作用:在处理继承问题上，更系统，更规范，定义抽象基类是为了给子类继承实现具体的功能，它是”残缺的类“，里面没有抽象方法的具体代码，里面的抽象方法是被子类重写的！

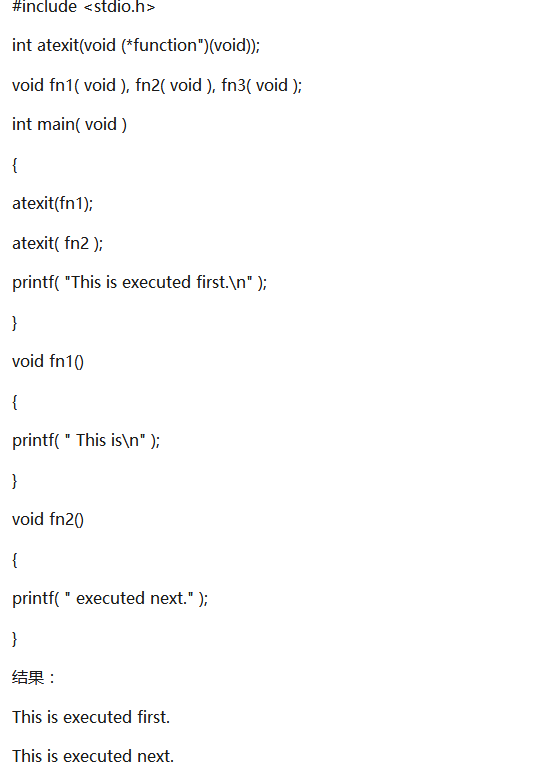


**有哪几种情况只能用构造函数初始化列表而不能用赋值初始化？**

const成员，引用成员

**main函数执行之前会执行什么？执行之后还能执行代码吗？**

全局变量、静态变量的空间分配和赋初值，全局对象的构造函数会在main函数之前执行，全局对象的析构函数在main函数之后执行；可以，可以用\_onexit 注册一个函数，它会在main 之后执行; 如果你需要加入一段在main退出后执行的代码，可以使用atexit()函数，注册一个函数。



**请描述进程和线程的区别？**

进程是资源的分配和调度的一个独立单元，而线程是CPU调度的基本单元；同一个进程中可以包括多个线程，并且线程共享整个进程的资源（寄存器、堆栈、上下文），一个进程至少包括一个线程。线程中执行时一般都要进行同步和互斥，因为他们共享同一进程的所有资源。

**TCP和UDP有什么区别。**

1.TCP协议在传送数据段的时候要给段标号；UDP 协议不需要。

2.TCP协议可靠；UDP协议不可靠。

3.TCP协议是面向连接；UDP协议采用无连接。

4.TCP协议负载较高,采用虚电路；UDP协议低负载。

5.TCP协议的发送方要确认接受方是否收到数据段(3次握手协议)。

6.TCP协议采用窗口技术和流控制。

**如何引用一个已经定义过的全局变量？**

可以用引用头文件的方式，也可以用extern关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变量，假定你将那个变量写错了，那么在编译期间会报错，如果你用extern方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错。

**多态类中的虚函数表是 Compile-Time，还是 Run-Time时建立的?**

虚拟函数表是在编译期就建立了,各个虚拟函数这时被组织成了一个虚拟函数的入口地址的数组。而对象的隐藏成员--虚拟函数表指针是在运行期--也就是构造函数被调用时进行初始化的，这是实现多态的关键。

**全局变量放在数据段，内部变量static int count；放在数据段，内部变量char \*p=“AAA”，p的位置在堆栈上，指向的空间的位置数据段，内部变量char  
\*p=new char；p的位置堆，指向的空间的位置数据段**

**C++函数中值的传递方式有哪几种?**

三种传递方式为：值传递、指针传递和引用传递。

**C++里面是不是所有的动作都是main()引起的？如果不是，请举例.**

比如全局变量的初始化，就不是由main函数引起的

举例： class A{};

A a; //a的构造函数限执行

int main() {}

**内联函数在编译时是否做参数类型检查？**

内联函数要做参数类型检查, 这是内联函数跟宏相比的优势。

**static全局变量与普通的全局变量有什么区别？static局部变量和普通局部变量有什么区别？static函数与普通函数有什么区别？**

这两者的区别在于非静态全局变量的作用域是整个源程序， 当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用， 因此可以避免在其它源文件中引起错误；从以上分析可以看出， 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围；static函数与普通函数作用域不同,仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。

**编译型语言和解释型语言区别？**

因为编译型语言是一次性的编译成机器码，所以可以脱离开发环境独立运行，而且通常**运行效率较高**；但因为编译性语言的程序被编译成特定平台上的机器码，因此编译生成的可执行程序通常**无法移植到其他平台上运行；**解释型语言是指使用专门的解释器对源程序进行逐行解释成特定平台的机器码并立即执行的语言。解释型语言通常不会进行整体的编译和链接处理，解释型语言相当于把编译型语言中的编译和解释过程混合在一起同时完成。但解释性语言有一个优势：跨平台比较容易，只需提供特定平台的解释器即可，每个特定平台上的解释器负责将源程序解释成特定平台的机器指令即可。如Java Python Ruby是解释型语言。

**预处理-编译-汇编-链接**

1）宏、文件包含、条件编译，以#开头的指令；#include指令就是一个预处理指令，它把头文件的内容添加到.cpp文件中还有其他许多预处理指令，可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉。

2）将预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后，产生相应的汇编代码文件。语法检查、类型检查

3）将编译完的汇编代码文件翻译成机器指令，并生成可重定位目标程序的.o文件，该文件为二进制文件，字节编码是机器指令。

4）**通过链接器将一个个目标文件（或许还会有库文件）链接在一起生成一个完整的可执行程序。** 链接程序的主要工作就是将有关的目标文件彼此相连接，也就是将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

**#pragma once #ifndef**

因为#ifndef受C/C++语言标准的支持，不受编译器的任何限制；而#pragma once方式却不受一些较老版本的编译器支持，一些支持了的编译器又打算去掉它，所以它的兼容性可能不够好; #ifndef不光可以保证同一个文件不会被包含多次，也能保证内容完全相同的两个文件（或者代码片段）不会被不小心同时包含, 缺点就是如果不同头文件中的宏名不小心“撞车”，可能就会导致你看到头文件明明存在，编译器却硬说找不到; **#pragma once 一般由编译器提供保证：同一个文件不会被包含多次**。注意这里所说的“同一个文件”是指物理上的一个文件，而不是指内容相同的两个文件。你无法对一个头文件中的一段代码作pragma once声明，而只能针对文件。其好处是，你不必再费劲想个宏名了，当然也就不会出现宏名碰撞引发的奇怪问题。大型项目的编译速度也因此提高了一些。

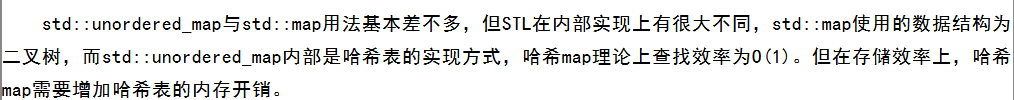
**C++11新特性**

<https://www.cnblogs.com/feng-sc/p/5710724.html>

在C99里面，NULL可以被定义为0或者0L（32位和64位的区别），或者直接就是由0或者0L转成的成void\*；在C++中，一个指向空的指针要么是一个字面值整形，要么是一个std::nullptr\_t







智能指针只是用对象去管理一个资源指针，同时用一个计数器计算当前指针引用对象的个数，当管理指针的对象增加或减少时，计数器也相应加1或减1，当最后一个指针管理对象销毁时，计数器为1，此时在销毁指针管理对象的同时，也把指针管理对象所管理的指针进行delete操作。

weak\_ptr与shared\_ptr最大差别在于赋值时不会引起智能指针计数增加，大众认为是为了解决shared\_ptr在相互引用的情况下出现的问题而存在的

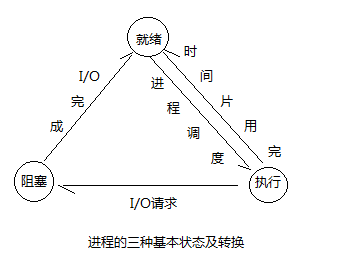
**TCP三次握手四次挥手**

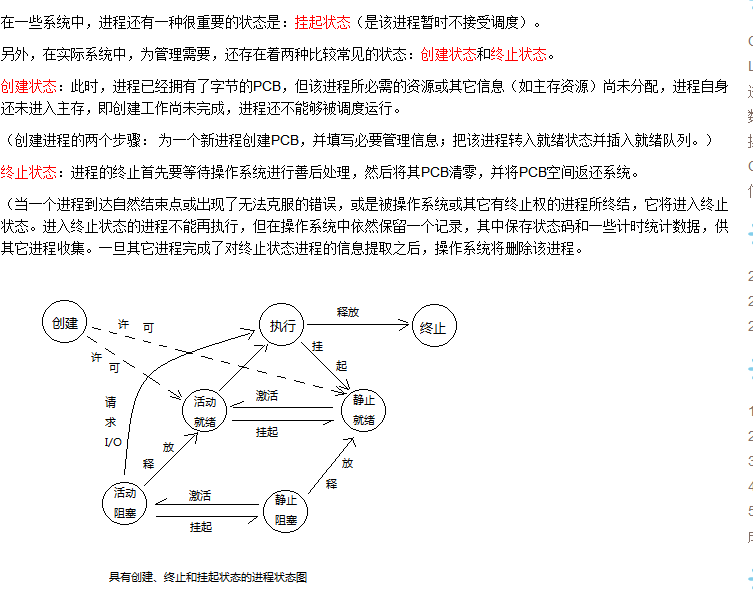
[**https://www.cnblogs.com/Andya/p/7272462.html**](https://www.cnblogs.com/Andya/p/7272462.html)

**进程三种状态**

就绪状态：进程已获得除CPU外的所有必要资源，只等待CPU时的状态。一个系统会将多个处于就绪状态的进程排成一个就绪队列。  
执行状态：进程已获CPU，正在执行。单处理机系统中，处于执行状态的进程只一个；多处理机系统中，有多个处于执行状态的进程。

阻塞状态：正在执行的进程由于某种原因而暂时无法继续执行，便放弃处理机而处于暂停状态，即进程执行受阻。（这种状态又称等待状态或封锁状态）通常导致进程阻塞的典型事件有：请求I/O，申请缓冲空间等。





**进程死锁**

[**https://blog.csdn.net/yanxiaolx/article/details/51944048**](https://blog.csdn.net/yanxiaolx/article/details/51944048)

产生死锁原因：1）竞争不可抢占性资源2）竞争可消耗资源3）进程推进顺序不当

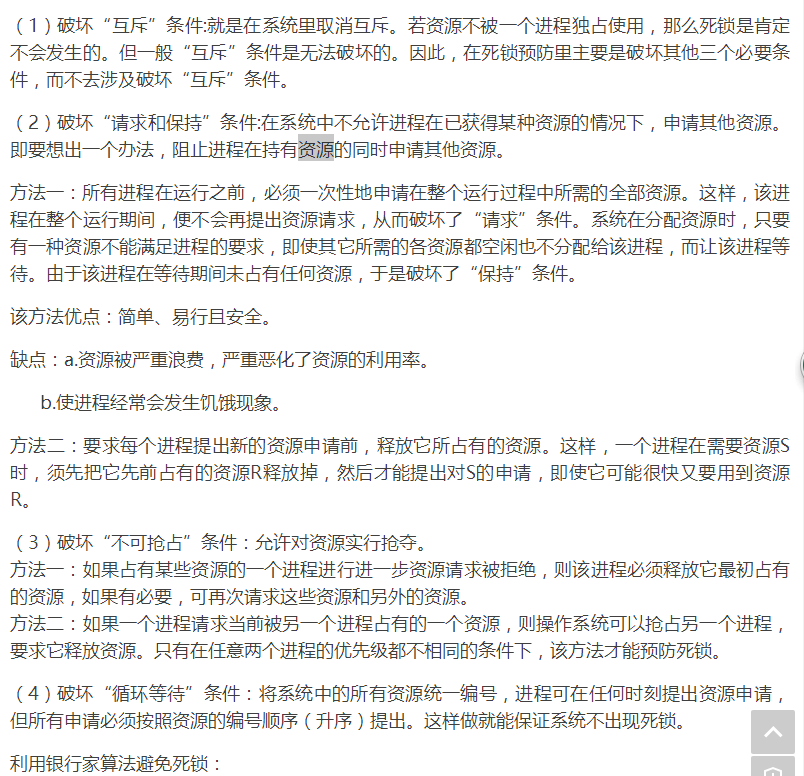
产生死锁必要条件：1）互斥条件：一个资源每次只能被一个进程使用。

2）请求和保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3）不可抢占条件:进程已获得的资源，在末使用完之前，不能强行剥夺，只能在进程使用完时由自己释放。

4）循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系

避免死锁方法：



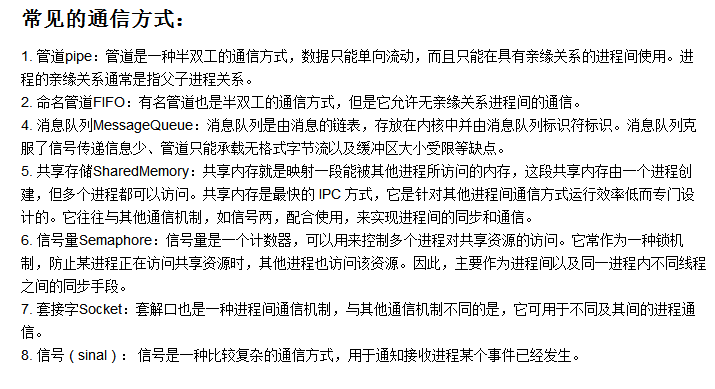
**进程饥饿**

有时资源分配策略可能是不公平的，即不能保证等待时间上界的存在。在这种情况下，即使系统没有发生[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81)，某些进程也可能会长时间等待．当等待时间给进程推进和响应带来明显影响时，称发生了进程饥饿，当饥饿到一定程度的进程所赋予的任务即使完成也不再具有实际意义时称该进程被饿死。举个例子，当有多个进程需要打印文件时，如果系统分配打印机的策略是最短文件优先，那么长文件的打印任务将由于短文件的源源不断到来而被无限期推迟，导致最终的饥饿甚至饿死。

**进程通信方式**

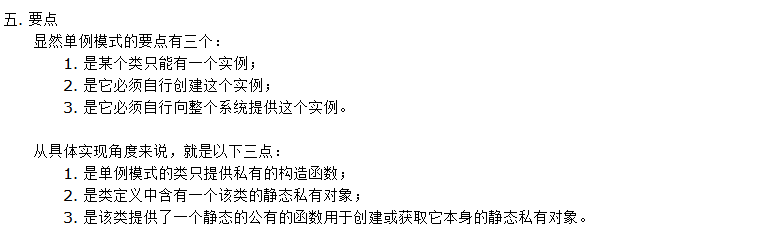
<https://www.cnblogs.com/LUO77/p/5816326.html>

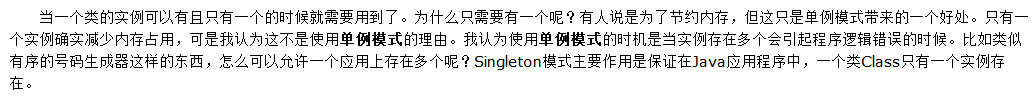
**信号、管道、消息队列、共享内存**

****

**单例模式**

<https://www.cnblogs.com/tjudzj/p/4450975.html>





**C++弱引用强引用**

[**https://www.cnblogs.com/kuzhon/articles/5648807.html**](https://www.cnblogs.com/kuzhon/articles/5648807.html)

weak\_ptr和shared\_ptr共享一个引用计数对象，在引用计数对象上增加一个weak\_count, 但不增加ref\_count.引用计数对象当ref\_count减至zero时会销毁其管理的资源，weak\_ptr可以通过ref\_count是否为0来判断指向的资源是否可用。当ref\_count和weak\_count都为0时引用计数对象会销毁其自身。

在多线程程序中，一个对象如果被多个线程访问，一般使用shared\_ptr，通过引用计数来保证对象不被错误的释放导致其他线程访问出现问题。

但这种引用计数解决不了循环引用的问题

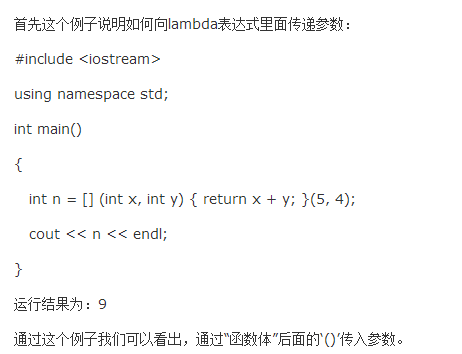
弱引用：它仅仅是对象**存在时候的**引用，当对象不存在时弱引用能够检测到，从而避免非法访问，弱引用也不会修改对象的引用计数。这意味这弱引用它并不对对象的内存进行管理，所以不能直接使用资源，weak\_ptr并没有重载-> 和 \* 操作符，所以我们不能通过他来直接使用资源，我们可以通过lock来获得一个shared\_ptr对象（shared\_ptr<int> sp2 = wp.lock()）。在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。进行该对象的内存管理的是那个强引用的boost::share\_ptr。boost::weak\_ptr只是提供了对管理对象的一个访问手段。boost::weak\_ptr除了对所管理对象的基本访问功能（通过get()函数）外，还有两个常用的功能函数：expired()用于检测所管理的对象是否已经释放；lock()用于获取所管理的对象的强引用指针。

智能指针详解

<https://blog.csdn.net/zsc_976529378/article/details/52250597>

**C++ lambda表达式**

[**https://www.cnblogs.com/alwayswangzi/p/7137427.html**](https://www.cnblogs.com/alwayswangzi/p/7137427.html)

****

**区别Increment/Decrement的前置/后置形式 （More Effective C++）**

前置和后置是指++/--符号。处理用户自定义类型时尽量使用前置，因为其不产生临时对象，原因如下。前置返回引用，后置会产生临时对象。

**前置式：Upint& operator++();后置 const Upint operator++(int)**

**Upint& operator++()**

**{ \*this+=1;**

**Return \*this**

**}**

**const Upint operator++(int){}**

**{ Upint oldvalue=\*this;**

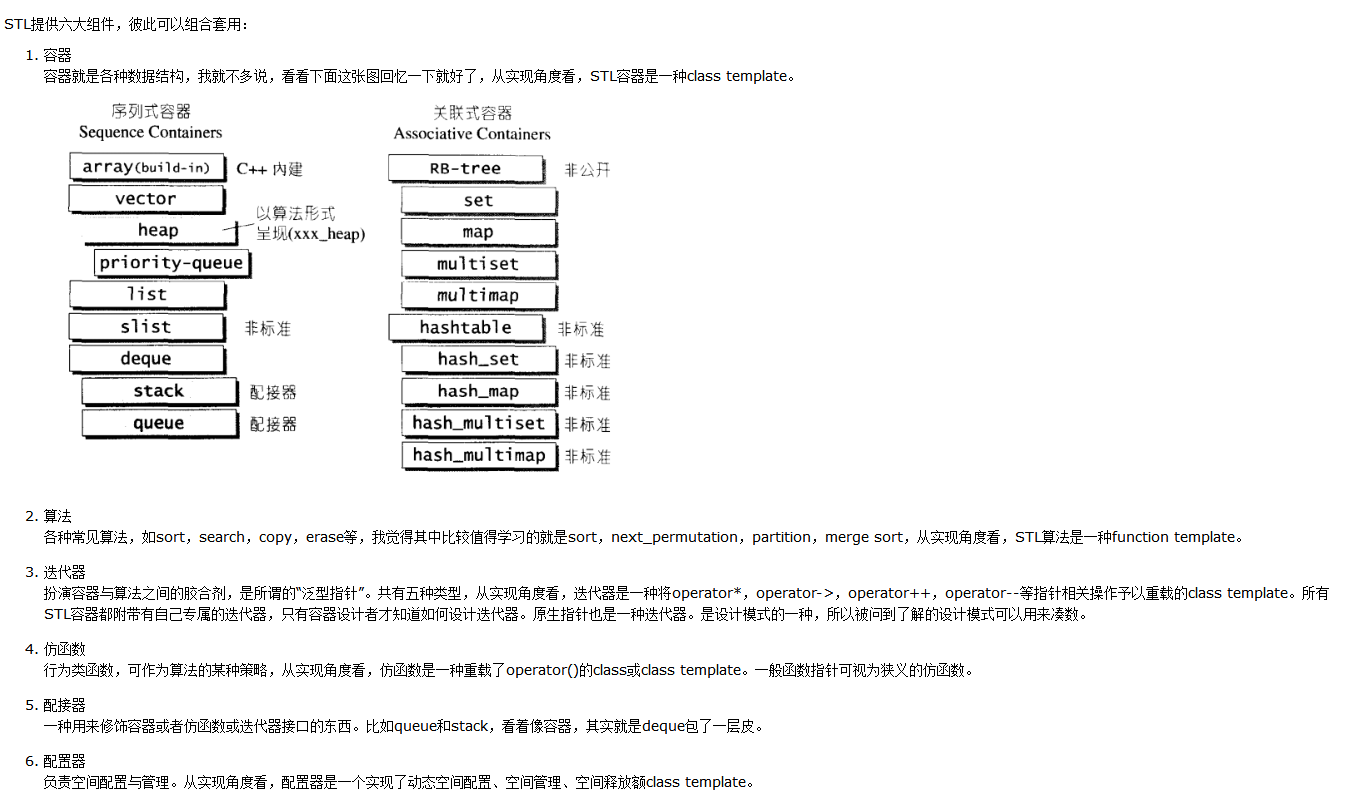
**++(\*this)**

**return oldvalue;**

**}**

**STL**

set/map的迭代器为const\_iterator类型，无法对元素进行修改（set的iterator为const的，而map的iterator的key为const的）。const vector<int>::iterator中，const是修饰的迭代器，也就是是个常迭代器，一旦初始化比如=a.begin()，再不能更改它的值，比如赋值=a.end()是不行的，递增递减操作等都不允许。虽然类似指针，但指针是内置类型，所以编译器可以通过const的位置来判断是常指针还是指向常量的指针，而迭代器只是一个对象，所以编译器不能分辨，所以用const\_iterator来取代指向常量的指针，使用它，你通过这个迭代器对迭代器所指向的内容进行改写是非法的。



vector内存用完，会以当前size大小重新申请2\*size的内存，然后把原来的元素复制过去，把新元素插上，然后释放原来的内存。一般我们释放vector里的元素使用clear，其实它不能释放内存，要想释放内存要使用swap。

<https://www.cnblogs.com/wulala1119/p/4758345.html>

**vector每次insert或erase之后，以前保存的iterator会不会失效？**

理论上会失效，理论上每次insert或者erase之后，所有的迭代器就重新计算的，所以都可以看作会失效，原则上是不能使用过期的内存  
但是vector一般底层是用数组实现的，我们仔细考虑数组的特性，不难得出另一个结论，

insert时，假设insert位置在p，分两种情况：  
a) 容器还有空余空间，不重新分配内存，那么p之前的迭代器都有效，p之后的迭代器都失效  
b) 容器重新分配了内存，那么p之后的迭代器都无效咯

erase时，假设erase位置在p，则p之前的迭代器都有效并且p指向下一个元素位置（如果之前p在尾巴上，则p指向无效尾end），p之后的迭代器都无效。

#### hash\_map和map的区别在哪里？

hash\_map底层是散列的所以理论上操作的平均复杂度是常数时间，map底层是红黑树，理论上平均复杂度是O(logn)，下面是借鉴的网上的总结：  
这里总结一下，选用map还是hash\_map，关键是看关键字查询操作次数，以及你所需要保证的是查询总体时间还是单个查询的时间。如果是要很多次操作，要求其整体效率，那么使用hash\_map，平均处理时间短。如果是少数次的操作，使用 hash\_map可能造成不确定的O(N)，那么使用平均处理时间相对较慢、单次处理时间恒定的map，考虑整体稳定性应该要高于整体效率，因为前提在操作次数较少。如果在一次流程中，使用hash\_map的少数操作产生一个最坏情况O(N)，那么hash\_map的优势也因此丧尽了。

#### 写多读少应该用什么容器？

私以为是链表，链表的插入操作时常数时间复杂度，访问操作是O(n)，是最适合写多读少的容器。

**reserve作用**

reserve是容器预留空间，提前预知需要存储元素数量可以避免不必要的内存重新分配带来的性能损失。但是要注意，如果容器中reserve的容量太大，而实际中根本不需要那么多，或者元素插入完成后，以后就不会再进行插入操作了，那么多出来的这部分空间就是过剩空间，而对于过剩空间，我们可以通过“交换技术”成功缩小空间，交换技巧的变体可以用于清除容器和减少它的容量到你的实现提供的最小值

**reserve和resize区别**

reserve是容器预留空间，但并不真正创建元素对象，在创建对象之前，不能引用容器内的元素，因此当加入新的元素时，需要用push\_back()/insert()函数。

resize是改变容器的大小，并且创建对象，因此，调用这个函数之后，就可以引用容器内的对象了，因此当加入新的元素时，用operator[]操作符，或者用迭代器来引用元素对象。

再者，两个函数的形式是有区别的，reserve函数之后一个参数，即需要预留的容器的空间；resize函数可以有两个参数，第一个参数是容器新的大小，第二个参数是要加入容器中的新元素，如果这个参数被省略，那么就调用元素对象的默认构造函数。

#### auto\_ptr可以做vector的元素吗？为什么？

不能。因为STL的标准容器规定它所容纳的元素必须是可以拷贝构造和可被转移赋值的。而auto\_ptr不能，可以用shared\_ptr智能指针代替。

**C++五种类型迭代器**

****

**STL 空间配置器**

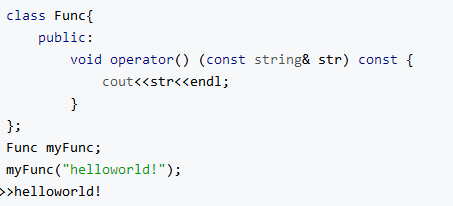
[**https://blog.csdn.net/hackbuteer1/article/details/7724534**](https://blog.csdn.net/hackbuteer1/article/details/7724534)

大多数情况下，STL默认的allocator就已经足够了。这个allocator是一个由两级分配器构成的内存管理器，当申请的内存大小大于128byte时，就启动第一级分配器通过malloc直接向系统的堆空间分配，如果申请的内存大小小于128byte时，就启动第二级分配器，从一个预先分配好的内存池中取一块内存交付给用户，这个内存池由16个不同大小（8的倍数，8~128byte）的空闲列表组成，allocator会根据申请内存的大小（将这个大小round up成8的倍数）从对应的空闲块列表取表头块给用户。

这种做法有两个优点：  
     (1)小对象的快速分配。小对象是从内存池分配的，这个内存池是系统调用一次malloc分配一块足够大的区域给程序备用，当内存池耗尽时再向系统申请一块新的区域，整个过程类似于批发和零售，起先是由allocator向总经商批发一定量的货物，然后零售给用户，与每次都总经商要一个货物再零售给用户的过程相比，显然是快捷了。当然，这里的一个问题时，内存池会带来一些内存的浪费，比如当只需分配一个小对象时，为了这个小对象可能要申请一大块的内存池，但这个浪费还是值得的，况且这种情况在实际应用中也并不多见。  
     (2)避免了内存碎片的生成。程序中的小对象的分配极易造成内存碎片，给操作系统的内存管理带来了很大压力，系统中碎片的增多不但会影响内存分配的速度，而且会极大地降低内存的利用率。以内存池组织小对象的内存，从系统的角度看，只是一大块内存池，看不到小对象内存的分配和释放。

**STL仿函数**

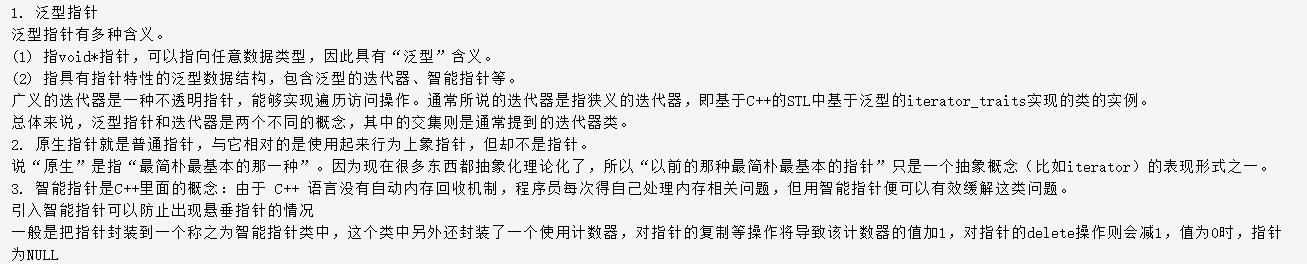
仿函数的主要用途是为了搭配STL算法，比如sort中的cmp函数，函数指针可以理解为一种狭义的仿函数。仿函数以操作数分为一元仿函数和二元仿函数，以功能分为算术运算、关系运算、逻辑运算。仿函数是一个能行使函数功能的类，语法几乎和普通函数调用一样，但是作为仿函数的类必须重载operator()运算符，如：



仿函数既能像普通函数一样传入给定数量的参数，还能存储或者处理更多我们需要的有用信息，因为它是一个类。如：



# [泛型指针，原生指针和智能指针](http://www.cnblogs.com/ransw/p/3955514.html)

****

原生指针即   (类型名\*p)样子的指针，类型名可以是基础类型，如int，double等，也可以是一个自己定义的Class类

相反的如果一个类重载了‘\*’和‘->’的运算符，可以像指针一样用‘\*’和‘->’操作，就不是原生的，如iterator等。

**模板全特化和偏特化**

****

****

**STL 配接器**

[**https://blog.csdn.net/lanchunhui/article/details/50934031**](https://blog.csdn.net/lanchunhui/article/details/50934031)

**Adapter：将一个class的接口转换为另一个class的接口，使原本因接口不兼容而不能合作的classes可以一起运作，一言蔽之，adapter用来改变接口，扮演着转换器的角色。**

* （1）改变仿函数（functors）接口，称之为 function adapter
* （2）改变容器（containers）接口，称之位 container adapter
* （3）改变迭代器（iterators）接口者，称之为 iterator adapter

****

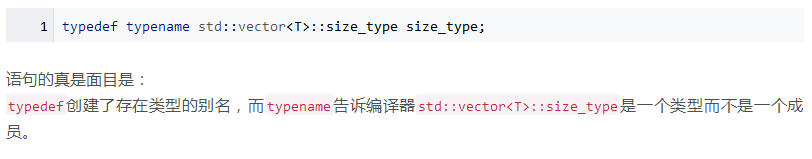
**STL traits（较为晦涩，主要牵涉到了内嵌和模板）**

[**https://blog.csdn.net/terence1212/article/details/52287762**](https://blog.csdn.net/terence1212/article/details/52287762)

STL的中心思想就在于：将数据容器和算法分开，彼此独立设计，最后再以一帖胶着剂将它们撮合在一起，也就是迭代器。利用traits可以获得迭代器属性，它利用内嵌类型和C++ template参数推导功能以及模板偏特化（针对原生指针）可将迭代器的属性萃取出来。

**iterator\_traits萃取机**

**前提知识：**

****

****

**STL 萃取机源码**

****

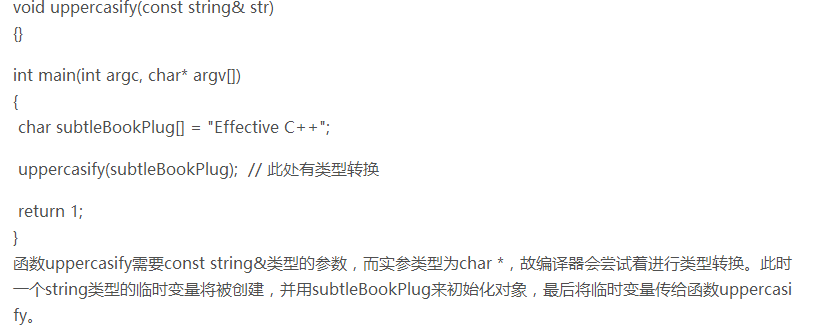
**dynamic\_cast**

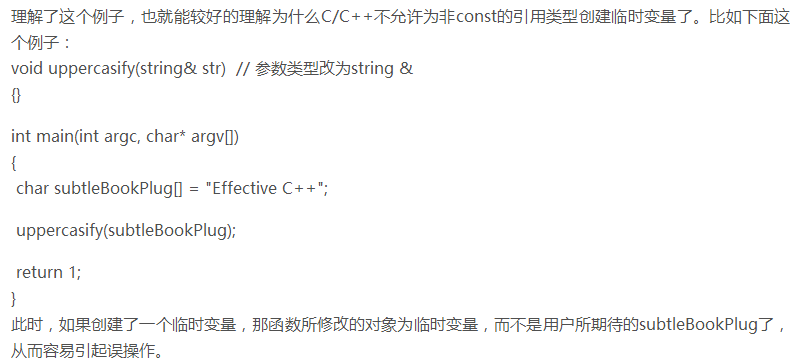
作为四个内部类型转换操作符之一的dynamic\_cast和传统的C风格的强制类型转换有着巨大的差别。除了dynamic\_cast以外的转换，其行为的都是在编译期就得以确定的，转换是否成功，并不依赖被转换的对象。而dynamic\_cast则不然。首先，dynamic\_cast依赖于**RTTI**信息，其次，在转换时，dynamic\_cast会检查转换的source对象是否真的可以转换成target类型，这种检查不是语法上的，而是真实情况的检查。先看RTTI相关部分，通常，许多编译器都是通过vtable找到对象的RTTI信息的，这也就意味着，如果基类没有虚方法，也就无法判断一个基类指针变量所指对象的真实类型, 这时候，dynamic\_cast只能用来做安全的转换,例如从派生类指针转换成基类指针.而这种转换其实并不需要dynamic\_cast参与.也就是说,dynamic\_cast是根据RTTI记载的信息来判断类型转换是否合法的.

在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

**临时变量**

临时变量通常在函数参数传递发生类型转换以及函数返回值时被创建，如，

****

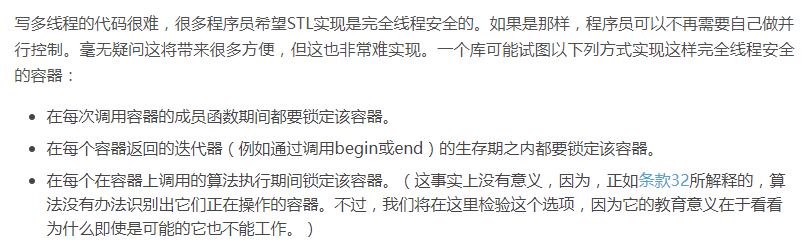
****

**右值引用**

**STL与线程安全**

[**https://blog.csdn.net/zdl1016/article/details/4161427**](https://blog.csdn.net/zdl1016/article/details/4161427)

[**http://blog.51cto.com/yaocoder/1208465**](http://blog.51cto.com/yaocoder/1208465)

****

****