1. 模板函数的原理！！ 我买的那本书上有

有个屁： 主要是指定特定的类型时才进行编译

1. 智能指针 shared\_ptr会出现相互引用的问题

主要是构造函数和赋值运算符重载的实现

Count用指针

2.四种类型转换

3.TCP UDP的区别 TCP的三次握手

4.虚函数的实现原理

对象最开始是虚函数指针，虚函数表中的第一项是类型？（可能有问题）用于typeid函数的调用，然后是虚函数指针，指向一个虚函数表，虚函数表是类所有的

5.malloc实现原理 ，看CSAPP 和侯捷内存管理视频 buddy系统和slab系统

Buddy系统：以页为单位分配，11个链表 最大页是1024。请求内存分配时，找对应的链表。如果不是2的幂次方时，剩下的挂到对应的链表中，找不到则去更大页的链表中找。释放时，归还到相应链表中，归还之前，如果相邻有空闲内存，进行合并

Slab以对象为单位，处理申请更小内存的情况（不够一个页）

Std:allocator:有个内存池的概念，也是不同大小的内存挂到不同的链表上，当申请内存很大时才用malloc，否则在空闲链表上寻找。

6.线程的经典问题 读写问题、生产者消费者问题

7.看下AC自动机和Trie树算法

8.蓄水池抽样法

周四面试准备

10.设计模式 策略模式、工厂模式、单例模式(周四上午看)、装饰者模式、观察者模式

11.项目准备

12.右值引用

刷leetcode 做头条笔试题！！！

360面试准备

1.select poll 原理

2.http原理

3.tcp ip协议

4.socket协议

IP地址 和端口

1.虚函数内部实现就是函数指针

2.strategy模式对性能提升有一定的帮助

3.Decorate模式的特点，其父类和构造器的参数是同一个类

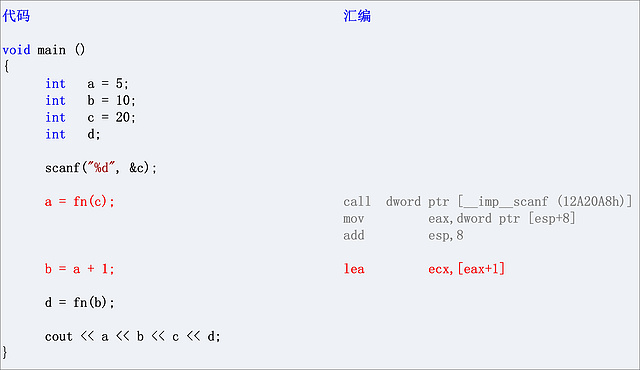
Using的使用:除了一般的命名空间外，还有以下用处：

1. 子类中调用父类的变量、函数(当子类private继承父类时，无法调用父类的变量，可以使用using)
2. 类似于typedef 别名，也可以对模板类进行别名（这是typedef无法做到的）

Vilotile

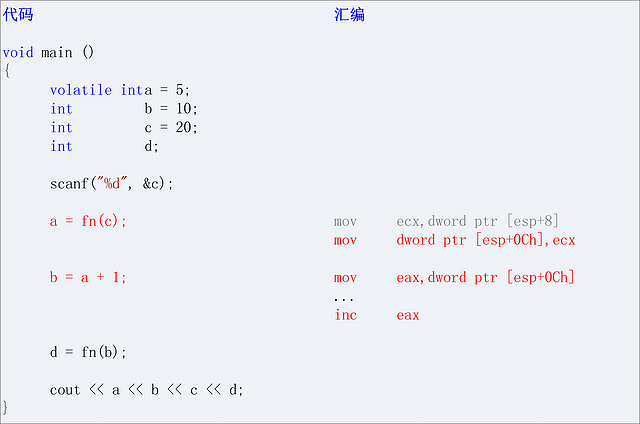
<https://www.cnblogs.com/god-of-death/p/7852394.html>

测试用例一：非Volatile变量

****

b = a + 1;这条语句，对应的汇编指令是：lea ecx, [eax + 1]。由于变量a，在前一条语句a = fn(c)执行时，被缓存在了寄存器eax中，因此b = a + 1；语句，可以直接使用仍旧在寄存器eax中的a，来进行计算，对应的也就是汇编：[eax + 1]。

**测试用例二：Volatile变量**

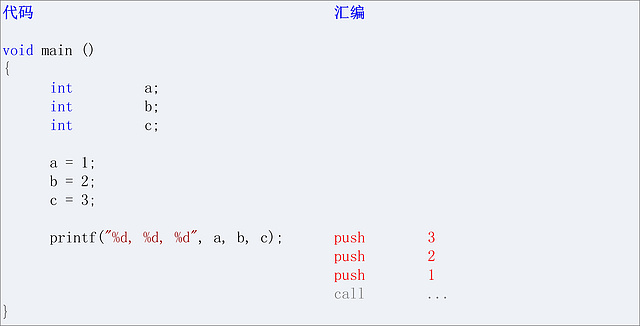


与测试用例一唯一的不同之处，是变量a被设置为volatile属性，一个小小的变化，带来的是汇编代码上很大的变化。a = fn(c)执行后，寄存器ecx中的a，被写回内存：mov dword ptr [esp+0Ch], ecx。然后，在执行b = a + 1；语句时，变量a有重新被从内存中读取出来：mov eax, dword ptr [esp + 0Ch]，而不再直接使用寄存器ecx中的内容。

**Volatile：不可优化的**

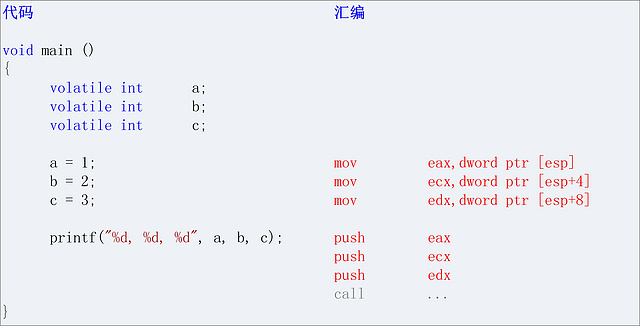
 与前面介绍的”易变”性类似，关于C/C++ Volatile关键词的第二个特性：”不可优化”性，也通过两个对比的代码片段来说明：

**测试用例三：非Volatile变量**



在这个用例中，非volatile变量a，b，c全部被编译器优化掉了 (optimize out)，因为编译器通过分析，发觉a，b，c三个变量是无用的，可以进行常量替换。最后的汇编代码相当简介，高效率。

**测试用例四：Volatile变量**



测试用例四，与测试用例三类似，不同之处在于，a，b，c三个变量，都是volatile变量。这个区别，反映到汇编语言中，就是三个变量仍旧存在，需要将三个变量从内存读入到寄存器之中，然后再调用printf()函数。

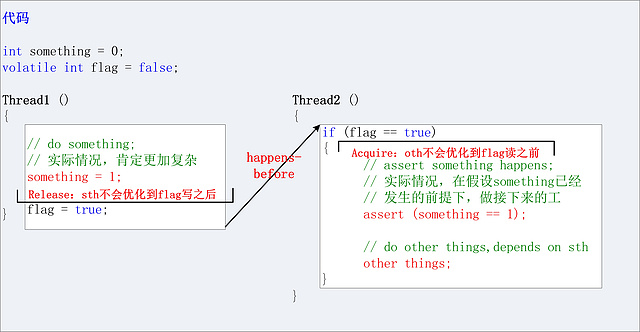
从测试用例三、四，可以总结出C/C++ Volatile关键词的第二个特性：“不可优化”特性。volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。相对于前面提到的第一个特性：”易变”性，”不可优化”特性可能知晓的人会相对少一些。但是，相对于下面提到的C/C++ Volatile的第三个特性，无论是”易变”性，还是”不可优化”性，都是Volatile关键词非常流行的概念。

C/C++ Volatile关键词前面提到的两个特性，让Volatile经常被解读为一个为多线程而生的关键词：一个全局变量，会被多线程同时访问/修改，那么线程内部，就不能假设此变量的不变性，并且基于此假设，来做一些程序设计。当然，这样的假设，本身并没有什么问题，多线程编程，并发访问/修改的全局变量，通常都会建议加上Volatile关键词修饰，来防止C/C++编译器进行不必要的优化。但是，很多时候，C/C++ Volatile关键词，在多线程环境下，会被赋予更多的功能，从而导致问题的出现

C/C++ Volatile关键词的第三个特性：”顺序性”，能够保证Volatile变量间的顺序性，编译器不会进行乱序优化。Volatile变量与非Volatile变量的顺序，编译器不保证顺序，可能会进行乱序优化。同时，C/C++ Volatile关键词，并不能用于构建happens-before语义，因此在进行多线程程序设计时，要小心使用volatile，不要掉入volatile变量的使用陷阱之中。

如此看来，C/C++ Volatile变量，与非Volatile变量之间的操作，是可能被编译器交换顺序的。

通过Java Volatile的Acquire、Release语义，对比C/C++ Volatile，可以看出，Java Volatile对于编译器、CPU的乱序优化，限制的更加严格了。Java Volatile变量与非Volatile变量的一些乱序操作，也同样被禁止。



在c++中，即使flag设置为volatile，编译器也可能将something 和flag的顺序打乱，所以也会出现先执行flag=true,再执行something=1，当执行到flag=true时，切换到Thread2，可能会引起Thread2崩溃

C++的Volatile关键词，有三个特性：易变性；不可优化性；顺序性

Deque的底层实现

C++模板函数的编译器处理：

<https://blog.csdn.net/abc_123_linbin/article/details/52931789>

当你不使用这个模版函数或模版类,编译器并不实例化它 ,当你使用时，编译器需要实例化它，

 因为编译器是一次只能处理一个编译单元, 也就是一次处理一个cpp文件,所以实例化时需要看到该模板的完整定义 .    所以都放在头文件中

  这不同于普通的函数, 在使用普通的函数时，编译时只需看到该函数的声明即可编译, 而在链接时由链接器来确定该函数的

其实模板实现不能放在cpp文件中，主要就是CPP在c++编译期间不能决定模板参数的类型，所以不能生成模板函数的实例，所以他会把模板类型带到链接期间，如果这个期间有函数调用了该实例，这个时候由于没有把模板实例到特定类型，就会导致编译错误。但是如果不调用该模板函数，那么是不会出现错误的。

右值引用：<https://www.jianshu.com/p/d19fc8447eaa>

很难得到左值和右值的真正定义，但是有一个可以区分左值和右值的便捷方法：**看能不能对表达式取地址，如果能，则为左值，否则为右值**。

它并不是重新分配一块新的空间，将要拷贝的对象复制过来，而是"偷"了过来，将自己的指针指向别人的资源，然后将别人的指针修改为nullptr，这一步很重要，如果不将别人的指针修改为空，那么临时对象析构的时候就会释放掉这个资源，"偷"也白偷了。下面这张图可以解释copy和move的区别。

如果我们没有提供移动构造函数，只提供了拷贝构造函数，std::move()会失效但是不会发生错误，因为编译器找不到移动构造函数就去寻找拷贝构造函数，也这是拷贝构造函数的参数是const T&常量左值引用的原因！

常量左值引用 既可以引用左值，也可以引用右值

emplace\_back()可以直接通过构造函数的参数构造对象，但前提是**要有对应的构造函数**。

移动语义对swap()函数的影响也很大，之前实现swap可能需要三次内存拷贝，而有了移动语义后，就可以实现高性能的交换函数了。

template <typename T>

void swap(T& a, T& b)

{

T tmp(std::move(a));

a = std::move(b);

b = std::move(tmp);

}

最大/小堆的构造：

构造最小堆，一定要从最后一个非叶子节点比较，每个非叶子节点进行下滤操作（也就是和当前节点的子节点进行比较）

构造函数可以写成虚函数吗：  
总之就是在调用构造函数之前，对象还没构建，也就没有对应的虚函数指针，所以也就不能调用虚的构造函数

虚函数调用是在部分信息下完成工作的机制，允许我们只知道接口而不知道对象的确切类型。 要创建一个对象，你需要知道对象的完整信息。 特别是，你需要知道你想要创建的确切类型。 因此，构造函数不应该被定义为虚函数。

从C++之父Bjarne的回答我们应该知道C++为什么不支持构造函数是虚函数了，简单讲就是没有意义。虚函数的作用在于通过子类的指针或引用来调用父类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过子类的指针或者引用去调用。

网络上还有一个很普遍的解释是这样的：虚函数相应一个指向vtable虚函数表的指针，但是这个指向vtable的指针事实上是存储在对象的内存空间的。假设构造函数是虚的，就须要通过 vtable来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以构造函数不能是虚函数。

本人对这个观点并不认同，这主要是因为用什么方式实现虚函数是编译器的事情，使用Vtable只是大多数编译器采用的一种手段，并不代表编译器实现不了虚构造函数，编译器之所以不支持虚构造函数主要原因就是没有必要，所以正好这种实现方式也不支持，巧合而已。

头条三面：

C++ weak\_ptr实现

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。

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.

wp是weak\_ptr，从weak\_ptr转成shared\_ptr

std::shared\_ptr<int> sp = wp.lock()

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.

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.

弱引用并不修改该对象的引用计数, 这意味这弱引用它并不对对象的内存进行管理. weak\_ptr 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。

注意: 虽然通过弱引用指针可以有效的解除循环引用, 但这种方式必须在程序员能预见会出现循环引用的情况下才能使用, 也可以是说这个仅仅是一种编译期的解决方案, 如果程序在运行过程中出现了循环引用, 还是会造成内存泄漏.

weak\_ptr与shared\_ptr继承同一个基类 \_Ptr\_base 因此有一样的资源指针字段、计数器字段。

<https://blog.csdn.net/qq_33762043/article/details/80072391> 实现

<https://blog.csdn.net/ypy9323/article/details/81154685>

其关键就是在加一个weak\_ptr的计数变量（最好和shared\_ptr的引用计数封装成一个struct），然后为了实现lock，在shared\_ptr中实现一个SharePtr(WeakPtr<T> const &w) 因为weak\_ptr都会关联一个shared\_ptr,这个函数也就相当于shared\_ptr的复制构造函数

Weak\_ptr中的复制构造函数：

WeakPtr(SharePtr<T>& s)

WeakPtr(WeakPtr<T>& w)

WeakPtr<T>& operator =(WeakPtr<T> & w)

WeakPtr<T>& operator =(SharePtr<T> & s) 这些函数都只增加weak\_ptr的引用计数，而不增加shared\_ptr的引用计数

bool expired() 在判断weak\_ptr的引用计数是否为0，一定要保证对应的shared\_ptr的引用计数为0，才能销毁对应的对象。

Unique\_ptr实现：

noexcept ：在C++11中，声明一个函数不可以抛出任何异常使用关键字noexcept.

1、为动态申请的资源提供异常安全保证

2、返回函数内动态申请资源的所有权

3、在容器中保存指针

4、管理动态数组

5、作为auto\_ptr的替代品

某个时刻只能有一个unique\_ptr 指向一个给定的对象。当它销毁时，它所指向的对象也会被销毁。

初始化 unique\_ptr 只能采用直接初始化的方式 （explicit 关键字）

不支持复制构造与赋值操作

在创建或者是reset一个具有删除器的unique\_ptr 时，必须提供删除器

不支持拷贝与赋值的规则有一个例外，那就是我们可以拷贝或者赋值一个将要被销毁的unique\_ptr（右值引用） ，比如：

unique\_ptr( unique\_ptr&& right\_value):

un\_ptr(right\_value.un\_ptr),del(std::move(right\_value.del)) {

right\_value.un\_ptr = nullptr ;

}

unique\_ptr& operator=( unique\_ptr&& right\_value ) noexcept {

if(this != &right\_value ){

std::cout << "operator && right\_value " << std::endl ;

del(\*this);

un\_ptr = right\_value.un\_ptr;

del = std::move(right\_value.del);

right\_value.un\_ptr = nullptr ;

}

return \*this ;

}

//u.release() u 放弃对指针的控制权，返回指针，并将 u 置为空

T\* release(){

T \*tmp = un\_ptr ;

un\_ptr = nullptr ;

return tmp ;

}

<https://blog.csdn.net/liushengxi_root/article/details/80672901>

哈希加密： 数据加盐

如何正确地存储用户的密码：<https://www.jianshu.com/p/cc2468b82e90>

##### 密码在磁盘上必须是以加密后的形式存在

有些童鞋可能对这个名词还有些陌生，这个盐当然不是我们平时吃的盐，也不是化学中的盐，而是一段字符串，用于和明文串接在一起然后哈希得到密文。比如我的密码是sunny (这当然不是真的密码...)，然后我的盐为abc，那么拼接在一起就可以是sunnyabc，然后以此哈希后存入数据库。当我再次登录的时候，再将我的密码加盐后哈希与数据库存储的密码比对。

盐（Salt）

在密码学中，是指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符，这种过程称之为“加盐”。

<https://blog.csdn.net/yang1982_0907/article/details/43483205>

是不是加了 Salt 之后就绝对安全了呢？淡然没有！坏人们还是可以他们数据字典中的密码，加上我们泄露数据库中的 Salt，然后散列，然后再匹配。但是由于我们的 Salt 是随机产生的，假如我们的用户数据表中有 30w 条数据，数据字典中有 600w 条数据，坏人们如果想要完全覆盖的坏，他们加上 Salt 后再散列的数据字典数据量就应该是 300000\* 6000000 = 1800000000000，一万八千亿啊，干坏事的成本太高了吧。但是如果只是想破解某个用户的密码的话，只需为这 600w 条数据加上 Salt，然后散列匹配。可见 Salt 虽然大大提高了安全系数，但也并非绝对安全。

客户端如何验证CA证书是可信任的

一般都是客户端验证服务端提供的证书是可信

CA证书与TLS介绍：<https://www.cnblogs.com/yxh168/p/9058043.html>

<https://blog.csdn.net/u011877584/article/details/81119441>

### 公钥传输威胁

   在A和B的通信中,C可以把自己的公钥发给A,让A把C的公钥当成B的公钥,这样的话.B拿到加密数据反而无法解密,而C却可以解密出数据.从而实现C截获AB之间的数据

   所以在两者的通信中必须要对公钥的来源进行确认

 A和B如果想安全交换公钥,就必须通过CA(证书颁发机构)  证书的通信过程        A和B首先都内置了CA的公钥

  根CA的证书是自己给自己签名的(自签名)

客户端和服务器端认证对方身份（依赖于PKI体系，利用数字证书进行 身份认证）,并协商通信中使用的安全参数、密码套件以及主密钥.

签名和加密：

我们说加密，是指对某个内容加密，加密后的内容还可以通过解密进行还原。 比如我们把一封邮件进行加密，加密后的内容在网络上进行传输，接收者在收到后，通过解密可以还原邮件的真实内容。

这里主要解释一下签名，签名就是在信息的后面再加上一段内容，可以证明信息没有被修改过，怎么样可以达到这个效果呢？一般是对信息做一个hash计算得到一个hash值，注意，这个过程是不可逆的，也就是说无法通过hash值得出原来的信息内容。在把信息发送出去时，把这个hash值加密后做为一个签名和信息一起发出去。 接收方在收到信息后，会重新计算信息的hash值，并和信息所附带的hash值(解密后)进行对比，如果一致，就说明信息的内容没有被修改过，因为这里hash计算可以保证不同的内容一定会得到不同的hash值，所以只要内容一被修改，根据信息内容计算的hash值就会变化。当然，不怀好意的人也可以修改信息内容的同时也修改hash值，从而让它们可以相匹配，为了防止这种情况，hash值一般都会加密后(也就是签名)再和信息一起发送，以保证这个hash值不被修改。至于如何让别人可以解密这个签名，这个过程涉及到数字证书等概念，我们后面在说到数字证书时再详细说明，这里您先只需先理解签名的这个概念。

为了向“客户”证明自己是“服务器”， “服务器”把一个字符串用自己的私钥加密，把明文和加密后的密文一起发给“客户”。对于这里的例子来说，就是把字符串 “你好，我是服务器”和这个字符串用私钥加密后的内容 {你好，我是服务器}[私钥|RSA] 发给客户。

“客户”收到信息后，她用自己持有的公钥解密密文，和明文进行对比，如果一致，说明信息的确是由服务器发过来的。也就是说“客户”把 {你好，我是服务器}[私钥|RSA] 这个内容用公钥进行解密，然后和“你好，我是服务器”对比。因为由“服务器”用私钥加密后的内容，由并且只能由公钥进行解密，私钥只有“服务器”持有，所以如果解密出来的内容是能够对得上的，那说明信息一定是从“服务器”发过来的。

在消息传输过程中，用公钥密码算法加密是不安全的。当客户端向服务端发送一个用公钥加密后的消息，服务端收到后用私钥解密，然后向客户端发送一个用私钥加密的消息，但是除了客户端，其他人都知道公钥，所以别人也能解密这个消息。所以一般把传输的消息进行对称加密。

总结一下，RSA加密算法在这个通信过程中所起到的作用主要有两个：

* **因为私钥只有“服务器”拥有，因此“客户”可以通过判断对方是否有私钥来判断对方是否是“服务器”。**
* **客户端通过RSA的掩护，安全的和服务器商量好一个对称加密算法和密钥来保证后面通信过程内容的安全。**

上述过程只能保证客户端可以确认服务器的身份，那么服务端如何把公钥发送给客户端呢？

但是这里还留有一个问题，在最开始我们就说过，“服务器”要对外发布公钥，那“服务器”如何把公钥发送给“客户”呢？我们第一反应可能会想到以下的两个方法：

a)把公钥放到互联网的某个地方的一个下载地址，事先给“客户”去下载。

b)每次和“客户”开始通信时，“服务器”把公钥发给“客户”。

但是这个两个方法都有一定的问题，

对于a)方法，“客户”无法确定这个下载地址是不是“服务器”发布的，你凭什么就相信这个地址下载的东西就是“服务器”发布的而不是别人伪造的呢，万一下载到一个假的怎么办？另外要所有的“客户”都在通信前事先去下载公钥也很不现实。

对于b)方法，也有问题，因为任何人都可以自己生成一对公钥和私钥，他只要向“客户”发送他自己的私钥就可以冒充“服务器”了。示意如下：

“客户”->“黑客”：你好           //黑客截获“客户”发给“服务器”的消息

“黑客”->“客户”：你好，我是服务器，这个是我的公钥    //黑客自己生成一对公钥和私钥，把公钥发给“客户”，自己保留私钥

“客户”->“黑客”：向我证明你就是服务器

“黑客”->“客户”：你好，我是服务器 {你好，我是服务器}[黑客自己的私钥|RSA]      //客户收到“黑客”用私钥加密的信息后，是可以用“黑客”发给自己的公钥解密的，从而会误认为“黑客”是“服务器”

为了解决这个问题，数字证书出现了，它可以解决我们上面的问题。先大概看下什么是数字证书，一个证书包含下面的具体内容：

* 证书的发布机构
* 证书的有效期
* 公钥
* 证书所有者（Subject）
* 签名所使用的算法
* 指纹以及指纹算法
* “客户”->“服务器”：你好
* “服务器”->“客户”：你好，我是服务器，这里是我的数字证书        //这里用证书代替了公钥
* “客户”->“服务器”：向我证明你就是服务器
* “服务器”->“客户”：你好，我是服务器 {你好，我是服务器}[私钥|RSA]
* **step1**： “客户”向服务端发送一个通信请求
* “客户”->“服务器”：你好
* **step2**： “服务器”向客户发送自己的数字证书。证书中有一个公钥用来加密信息，私钥由“服务器”持有
* “服务器”->“客户”：你好，我是服务器，这里是我的数字证书
* **step3**： “客户”收到“服务器”的证书后，它会去验证这个数字证书到底是不是“服务器”的，数字证书有没有什么问题，数字证书如果检查没有问题，就说明数字证书中的公钥确实是“服务器”的。检查数字证书后，“客户”会发送一个随机的字符串给“服务器”用私钥去加密，服务器把加密的结果返回给“客户”，“客户”用公钥解密这个返回结果，如果解密结果与之前生成的随机字符串一致，那说明对方确实是私钥的持有者，或者说对方确实是“服务器”。
* “客户”->“服务器”：向我证明你就是服务器，这是一个随机字符串     //前面的例子中为了方便解释，用的是“你好”等内容，实际情况下一般是随机生成的一个字符串。
* “服务器”->“客户”：{一个随机字符串}[私钥|RSA]
* **step4**： 验证“服务器”的身份后，“客户”生成一个对称加密算法和密钥，用于后面的通信的加密和解密。这个对称加密算法和密钥，“客户”会用公钥加密后发送给“服务器”，别人截获了也没用，因为只有“服务器”手中有可以解密的私钥。这样，后面“服务器”和“客户”就都可以用对称加密算法来加密和解密通信内容了。
* “服务器”->“客户”：{OK，已经收到你发来的对称加密算法和密钥！有什么可以帮到你的？}[密钥|对称加密算法]
* “客户”->“服务器”：{我的帐号是aaa，密码是123，把我的余额的信息发给我看看}[密钥|对称加密算法]
* “服务器”->“客户”：{你好，你的余额是100元}[密钥|对称加密算法]
* …… //继续其它的通信

注意，对称加密用到的密钥是由客户端用公钥加密后发送给服务端的

问题：

1、服务器”随随便便用私钥去加密一个来路不明的字符串并把结果发送给对方是不安全的。

“服务器”不直接加密收到的字符串，而是加密这个字符串的一个hash值，这样就避免了加密那些有规律的字符串，从而降低被破解的机率。“客户”自己发送的字符串，因此它自己可以计算字符串的hash值，然后再把“服务器”发送过来的加密的hash值和自己计算的进行对比，同样也能确定对方是否是“服务器”。

2、在双方的通信过程中，“黑客”可以截获发送的加密了的内容，虽然他无法解密这个内容，但是他可以捣乱，例如把信息原封不动的发送多次，扰乱通信过程。

可以给通信的内容加上一个序号或者一个随机的值，如果“客户”或者“服务器”接收到的信息中有之前出现过的序号或者随机值，那么说明有人在通信过程中重发信息内容进行捣乱，双方会立刻停止通信。

3、在双方的通信过程中，“黑客”除了简单的重复发送截获的消息之外，还可以修改截获后的密文修改后再发送，因为修改的是密文，虽然不能完全控制消息解密后的内容，但是仍然会破坏解密后的密文

在每次发送信息时，先对信息的内容进行一个hash计算得出一个hash值，将信息的内容和这个hash值一起加密后发送。接收方在收到后进行解密得到明文的内容和hash值，然后接收方再自己对收到信息内容做一次hash计算，与收到的hash值进行对比看是否匹配，如果匹配就说明信息在传输过程中没有被修改过。如果不匹配说明中途有人故意对加密数据进行了修改，立刻中断通话过程后做其它处理。

STL中不同容器的使用场景：