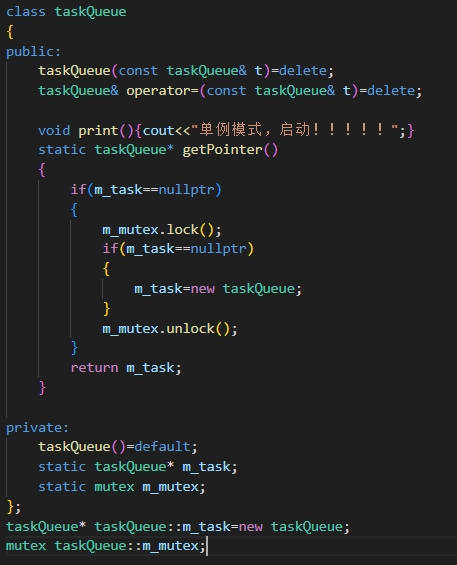
接下来看一下单例模式中的一种区分方法，这种方法将单例模式区分为了（懒汉模式）（饿汉模式）。其实这俩者没什么特别的，其实就是我们常见的是否懒加载的一种区分罢了。对于懒汉模式，一般不会在初始化时就进行new一个对象，而是指定一个空指针，等到存在一个具体的指针想要用到这个对象时我们才会去new一个出来。而饿汉模式就是在初始化时就直接对这个对象进行创建。

同时我们来考虑一些别的东西，我们知道懒汉模式是一种懒加载，饿汗模式是直接创建，那么这里其实是会在多线程时可能出现一些问题的。对于多线程下的饿汉模式，由于是已经创建了一个实例对象的，我们去取到对象是不会导致一个重复的内存申请的。因此它是一个线程安全函数。而如果是懒汉模式呢，我们是在调用静态公有函数时才去创建函数的，照我们CSAPP的学习，有没有一种可能，在我们调用这个函数时，发生了一个上下文切换，且同时切换到了一个申请语句，这样的话导致了我们想要去分配一个已经被分配的内存而导致程序崩溃。所以说，简单的new而不加任何保护的懒汉模式是线程危险的。同时，也是会破坏单例模式的性质的。

但是，我们其实也可以想到这种危险的解决办法，就是加锁嘛，接下来讨论这个。

现在我们来看使用**双重检查锁定**的方法来进行懒汉模式下的单例模式的线程安全问题解决。

算了，直接上代码，就能够理解了。



首先，我们通过一对互斥锁将m\_task的示例锁起来以防止线程冲突，这样的话确实能避免线程安全问题，但问题是这样的话效率太低了。分析一下，当一个线程获得一个锁后，其它的所有线程会进行等待，直到锁被释放，以此类推。而且我们来分析下这几个内层的性能消耗，每次进入都会进行一次加锁和解锁操作，至于判断和创建的，是不可避免的，不纳入考虑中。

这里我们需要考虑这里加锁和解锁操作，我们如果是单层检查锁定的话，每次都要进行加锁与解锁的操作，而这个是相当耗时的。这个会将懒汉模式相较于饿汉模式的性能优势丢失掉大半。因此，我们需要考虑通过一些操作来进行这里的锁操作的简化。这里就是考虑直接在外层再套一层条件判断，就是图中的成品。

在这种设计下，假设啊，先假设，所有的线程在运行时是同步到调用函数这一步，且切换到一个线程后该线程至少完整的执行了这个函数。那么我们来看。当一个进程运行完后，对应的m\_task被实例化，接着切换到下一个线程，先进行外层的条件判断，由于，已经实例化，所以这里会直接跳转到返回语句，不会再进行加锁和解锁的操作。这样就很大程度上解决了线程安全问题。

但是，我们也不难想到，这样的设计在极端情况喜爱还是可能存在问题的，就是多个线程都运行到了条件判断的内部，但是这种说实话真的太极端了。但是这种情况还是不能够忽略的，所以这里我们就需要来进入下一步。

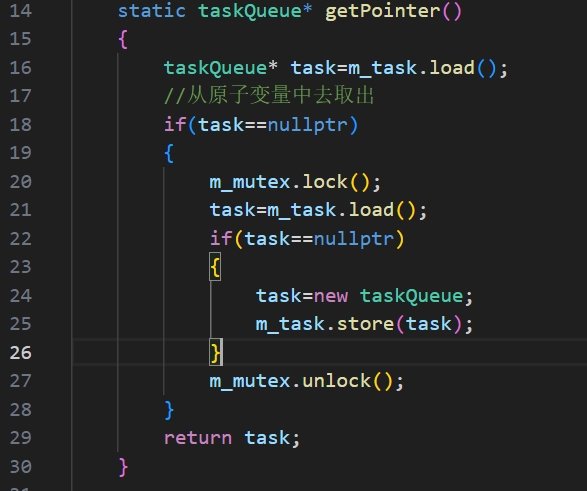
我们在CSAPP中已经知道了底层程序到底是怎么动的，其实对于程序来说，这里的代码顺序可能会因为编译器的奇怪优化而被重排顺序。这里关注到这里的new一行的操作。正常来说，我们期望的顺序应该是先去new一块堆区的内存，此时的内存是空白的。接着程序应该在这块内存中进行操作，去进行各个对象的初始化等创建操作。最后，程序应该把这块的内存地址送给我们的m\_task指针。此时就完成了一个new操作。

好，现在我们停下来，我们应该了解，对应的new操作对应的是一系列的汇编码。编译器的优化会导致这一系列的汇编码的顺序不可预测，就比如一种情况，程序先去申请一块内存，然后先将指针送给了对象，最后再调用构造函数进行初始化。这样是一种可能的情况，在单线程的情况下，这种重排是不会出现问题的。但是，我们考虑在多线程呢。如果一个线程在重排后的第二步，也就是指针赋值的这一步中进行了上下文切换，那么此时指针指向一块确实存在但并未初始化的值。此时接下来的线程如果也进入了这个函数，那么由于此时指针确实是存在地址的，那么程序将会直接从这个函数返回。而且由于这块地址是还未初始化的，那么这个线程接下来的所有有关该块内存区的操作将是非法的。直到线程重新切回之前那个卡在第二步的，知道其完成第三步其他线程才能正确执行。但是，一般来说，等不到那个时候，程序已经崩溃了。

为了解决这个问题，由于我们无法去确定编译器的行为，而且，不同电脑对于代码的优化是不尽相同的，所以想要去指令汇编指令的生成顺序是不现实的。你自己搓那你牛逼。所以这里我们需要使用到一个相对高抽象程度的特性，就是C++11中的原子变量。这个其实也是老朋友了。

C++中提供的原子变量是**atomic**。这个变量可以用来控制使用这个变量的相对于变量在底层中机器代码的执行顺序。如果我们将一个数据储存到原子变量中，那么这个数据对应的机器代码执行顺序我们就可以通过这个原子变量来进行指定。

对于一个原子变量，我们可以通过store方法来进行一个变量的塞入，可以用load方法来进行一个变量的取出。同时，我们还可以指定对应的执行顺序到底是一个什么样的，存在一个默认方法就是顺序执行。



将一个变量储存在原子变量中，再取出这个原子变量所管理的元素的话，这个元素就能够保证是以指定的顺序进行运行的。来分析一下这里的使用原子变量的版本。首先进入函数体后，使用load方法将对应的原子变量管理的取出。接着进行第一层判断，假设我们这里存在多个线程都完成了取出的这一步，然后会在条件判断这一块进行上下文切换的。此时所有的线程都会被阻塞在互斥锁这一行上。第一个碰到的线程将会取得互斥锁，其他的将会被阻塞。

接下来需要进行第二次加载，此时第一次拿到锁的对应的变量将是一个空指针，因此会进入内层循环，接着由于原子变量的属性不会对指令进行重排。所以接下来的new操作将会以申请内存，初始化(调研构造函数)分配指针的顺序进行。请注意，所谓的原子属性不会使得时间片延长，所以这里时间片到期了还是会进行线程的切换，但是其他的所有线程都会被锁给阻塞而不会进入这一层所以可以视为一种连续的操作。接着第一个取到锁的程序一直跑直到锁被释放。请注意，这里还是可能进行上下文切换，我们不能够假设这里一定会运行到整个函数结束不被打断。我们能假设的只有在整个锁中的执行不会被打断。

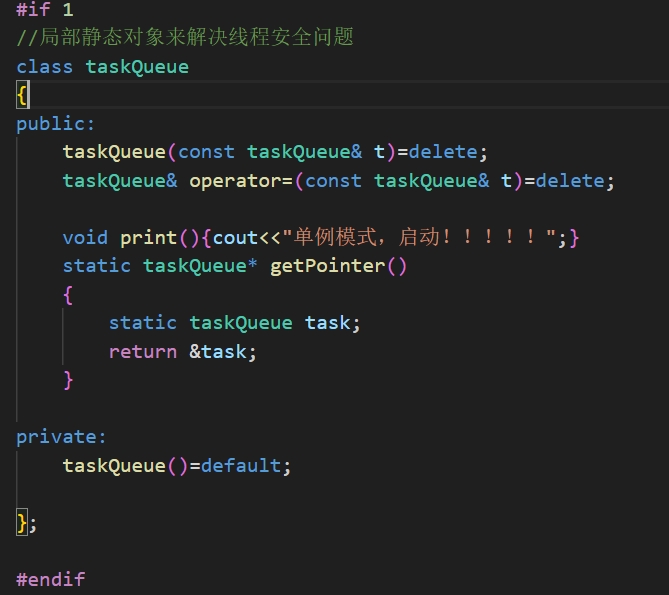
好，此时当第一个锁被释放后，程序的其他线程将能够获取对应的锁，此时拿到锁的线程将能继续往下跑，前面已经假设这些线程都已经跑过了第一层判断。那么这里的lock方法就体现了它的作用。这里的lock其实起到了一个更新的作用。如果这里没有这个函数，那么这里将会导致一种在已经初始化的情况下再视图进行一次初始化，这里就是双重检查的目的。那么由于这个更新，这个非第一个获得锁的线程将会直接条状到返回语句。并不会导致前面那种不依靠原子变量可能存在的线程安全问题。

除了使用双重检查锁定+原子变量来进行单例模式的设计，我们还可以使用别的一种方式来进行等效的设计，这个就是局部静态对象。而这个是C++11的特性，所以至少编译器需要支持C++11的版本。

这个解决方法不需要原子变量，也不需要互斥锁。

首先来看一下这里局部变量能够实现功能的技术基础。这个是依赖于C++11的一个新特性的

**如果指令进入一个未被初始化的声明变量，所有并发执行需要等待这个变量完成初始化。**



简单分析下，我们在getPoint函数中直接去创建一个静态类对象，这里缺少一个初始化操作，之后再补。就当其他的线程运行到这个函数时，由于C++11的特性，其他所有线程都被阻塞，知道第一个到达的线程去进行对应的初始化。当系统切换到其他的线程后这些个线程才能继续往下走。这是利用C++11的新特性来进行解决懒汉模式下的线程安全问题的一种方法，而且这种方法相对于前面的要使用原子变量和互斥锁的简单许多，设计上简单，而且一般来说效率也更高。