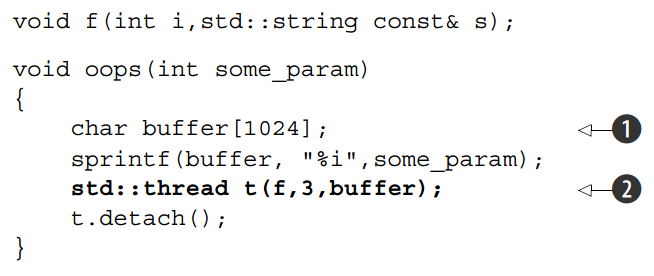
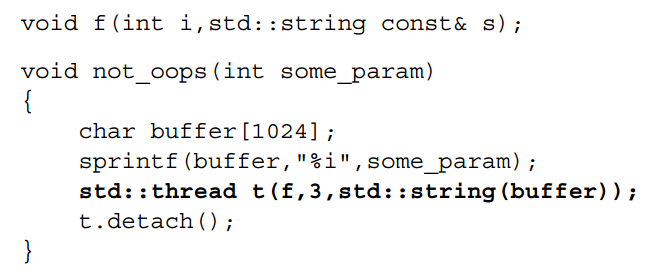
重要知识点：

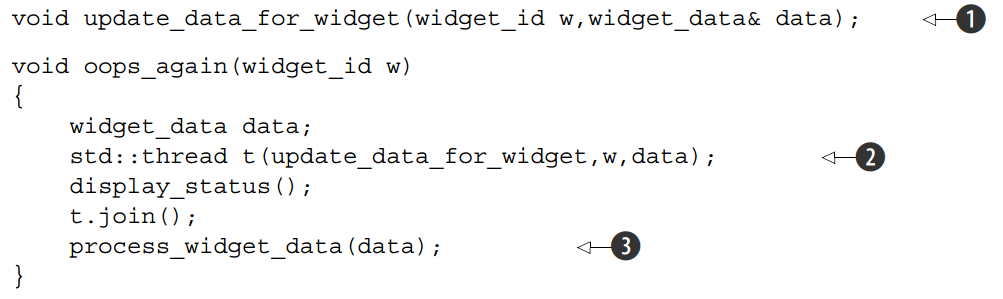
1. **thread在销毁前必须要调用detach或join方法。**
2. **thread的参数如果会发生隐式转换，这个过程是在线程中做的，所以需要注意作用域。**



图中1是个局部变量，在t开始执行时很有可能buffer已经被销毁，导致线程崩溃，正确方案应该是：



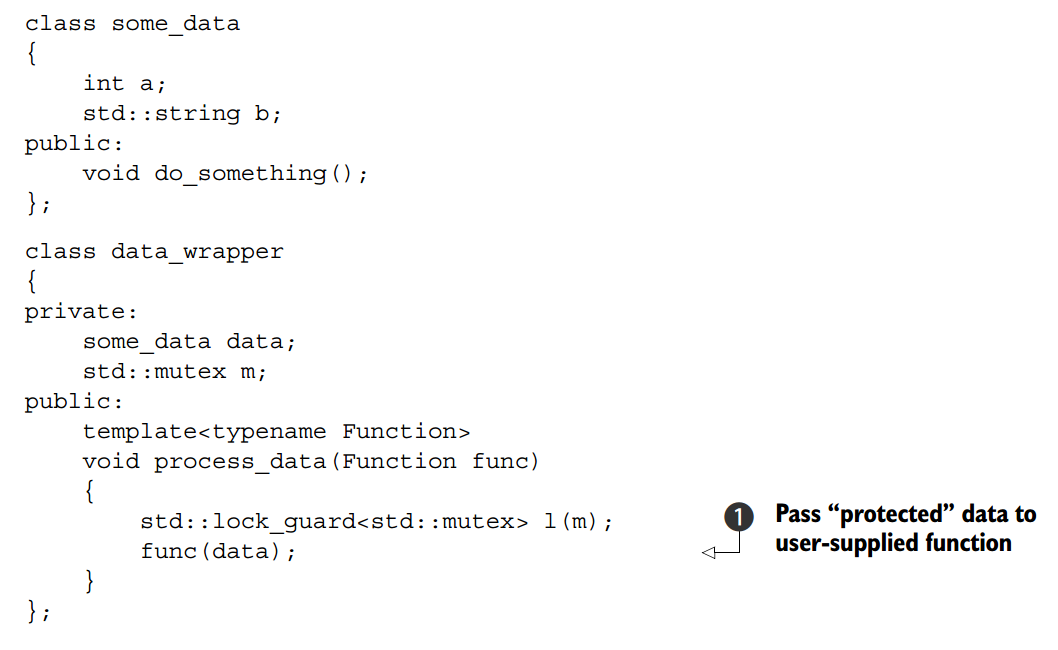
1. **thread的参数如果是引用，必须加上std::ref。**

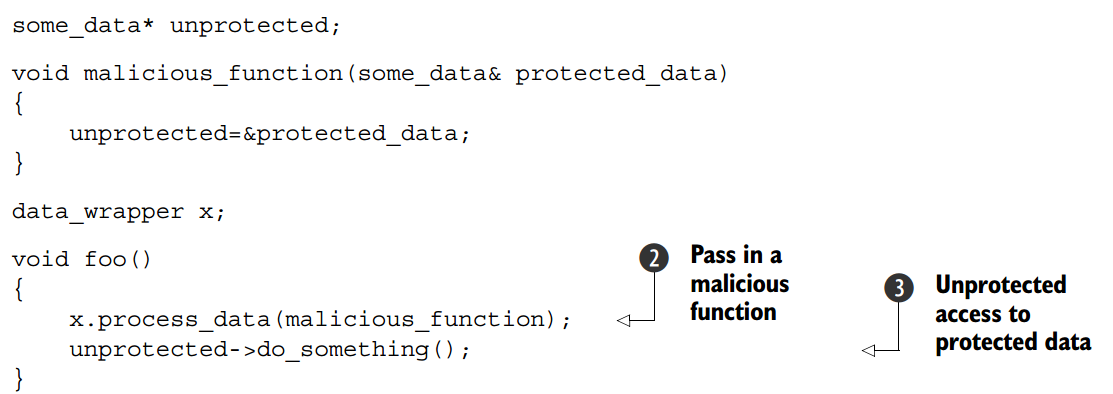


因为thread的构造函数中的变参是不认识引用的，所以必须加上std::ref来显式表示：

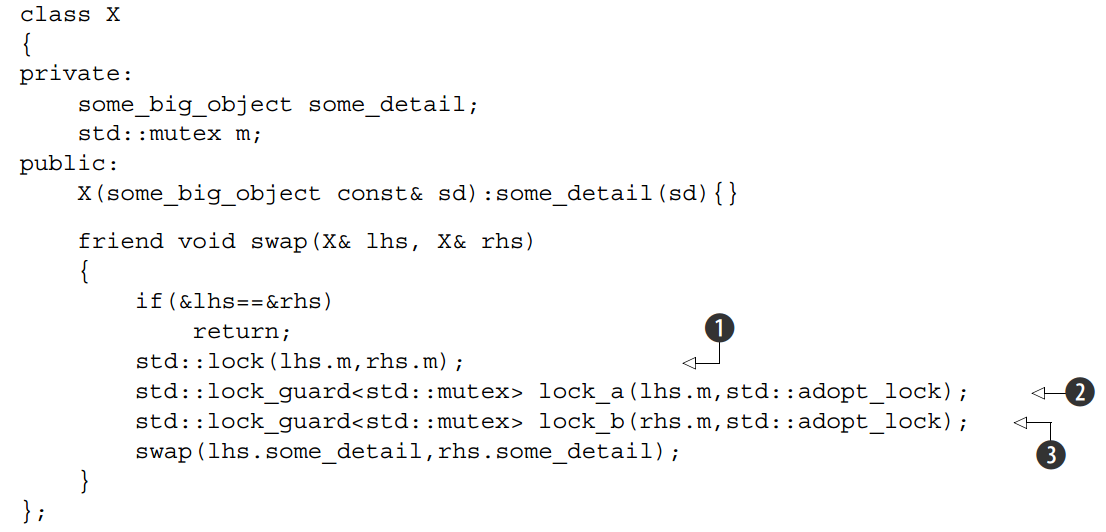


1. **thread无法copy，但是可以move。**
2. **Software Transactional Memory，大致是一种无锁同步的思想，类似数据库的事务，详情之后再研究，马克一下。**
3. **即使利用互斥锁来保护读写，但是如果不慎将内部数据的指针或引用返回出去的话，那么就可以避开互斥锁来对数据进行修改，这点务必需要注意。**





1. **std::lock提供了All-Or-Nothing的语义，可以同时对多个mutex上锁，避免死锁。**

****

其中std::adopt\_lock是为了告诉lock\_guard的mutex已经上过锁了。

1. **死锁还可能出现在两个thread相互在对方的thread中join。**
2. **避免嵌套的锁结构；避免在上锁的时候调用用户层的代码；给多个锁定优先级并永远按照优先级上锁；使用层次锁。**
3. **在RAII语义中，unique\_lock和lock\_guard类似，unique\_lock可以利用std::defer\_lock延迟上锁，lock\_guard可以利用std::adopt\_lock不对mutex上锁；lock\_guard快速轻巧但是安全性较低，而unique\_lock牺牲了性能空间换来了更高的安全性，因为unique\_lock在销毁的时候会检查它是否占用了这个lock来决定是否解锁。**
4. **上锁的粒度如果控制得不好，会导致性能问题。**
5. **存在std::condition\_variable和std::condition\_variable\_any两种条件变量，前者只适用于std::mutex而后者适用于任意mutex或mutex-like，因此前者效率更高而后者更通用。**
6. **如果一个条件变量被唤醒时发现它的条件仍然不满足，这种现象称之为spurious wakeup。**
7. **如果一个条件变量会频繁地发生spurious wakeup，说明这个时候并不适合使用condition\_variable做同步机制，而应该选择future。**