Subject: Re: Use same variable in different threads Posted by Didier on Mon, 13 Dec 2010 21:48:03 GMT View Forum Message <> Reply to Message

Hi Koldo,

I would add just one last thing: DEADLOCKs ==> what MT programs dread the most.

In practice, there is a case in C++ where deadlocks can appear without notice : when an exception occurs.

Imagine an exception occurs in the middle of you're "slow operation" ==> the mutex doesn't get released ==> a deadlock will come up soon enough

To avoid this I use, what I call a ScopedLock class:

```
template<class MUTEX>
class ScopedLock
{
private:
MUTEX& mutex;
private:
// The following constructor/operators are expilictly FORBIDDEN
    // because they have no meaning
ScopedLock(void) {};
ScopedLock(const ScopedLock& ) {};
ScopedLock& operator=(ScopedLock&) { return *this; };
public:
inline ScopedLock(MUTEX& mut)
: mutex(mut)
{
 mutex.lock();
}
inline ~ScopedLock(void)
{
 mutex.unLock();
}
};
```

The point is to create a 'ScopedLock' object when interring a protected zone of code, and when the scope ends ==> the unlock is automatically done IN ALL POSSIBLE CASES !! even exceptions: The compiler handles all for you

so you're code would become:

```
void ThreadFunction(){
for(int i = 0; i < 10; i++){
Thread::Sleep(Random(10)); // Pretend some work...
{
    ScopedLock(data.m);// Enter the section that accesses the shared data
    data.a = i;
    data.c << data.a << "\n";
Thread::Sleep(20); //Let's pretend that some slow operation happens here (for example file
    access)
    data.b = data.a;
} // implicit release
}</pre>
```

You don't have any more "mutex leaks" possible

```
Page 2 of 2 ---- Generated from $U$\sc ++$\sc Forum$
```