

Re: Share .cpp and .h along projects

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.language/2007-08/msg00590.html>

- *From:* "Ben Voigt [C++ MVP]" <rbv@xxxxxxxxxxxxxx>
 - *Date:* Thu, 16 Aug 2007 08:41:35 -0500
-

"Doug Harrison [MVP]" <dsh@xxxxxxx> wrote in message
news:qm87c3li5ota3g5bif09vopna7942vru98@xxxxxxxxxxx

On Wed, 15 Aug 2007 08:32:27 -0500, "Ben Voigt [C++ MVP]"
<rbv@xxxxxxxxxxxxxx> wrote:

#5 at least is a sign of code with real bugs in
it.

Oh, please. As I mentioned, being able to suppress
interprocedural
optimization is a necessity for implementing mutex
lock/unlock
operations,

You use the volatile keyword for that, then you are robust against future
improvements in the optimizer.

and it also eliminates one of the problems in the DCLP. By
putting
WaitForSingleObject, ReleaseMutex, and others in opaque
system DLLs,
correct compiler behavior for MT programming WRT these
operations
essentially comes for free.

Again, you use the volatile keyword for variables that are volatile.
Trying
to get volatile behavior by changing to DLLs is bad.

Uh, no.

Re: Share .cpp and .h along projects

The "volatile" keyword has essentially no use in multithreaded programming, although MS defining it to have memory barrier semantics in VC2005 makes its misuse more likely to work on systems with weakly ordered memory systems. It also provides expert users with a tool they can use non-portably, for those occasional times when it provides something of value.

The mutex lock/unlock situation I've been talking about is illustrated by this:

```
mutex mx;
int x;

// None of these touch x.
void lock(mutex&);
void unlock(mutex&);
void g(int);

void f1()
{
    lock(mx);
    g(x);
    unlock(mx);

    lock(mx);
    g(x);
    unlock(mx);
}

void f2()
{
    lock(mx);
    ++x;
    unlock(mx);
}
```

A compiler that can see into all these functions will observe that none of lock, unlock, and g access x, so it can cache the value of x across the mutex calls in f1. This of course is incorrect for multithreaded programming. In addition, the compiler could potentially move the initial read of x in front of f's first critical section. Putting lock/unlock into an opaque DLL suppresses this optimization, because the compiler must assume that all globals are reachable from functions it can't see into. If the compiler could look into the DLL, there would have to be some way to explicitly indicate that lock/unlock are unsafe for this optimization.

That is exactly what "volatile" does. Even in old versions of the compiler, when memory barriers are not used, volatile exactly prevents the optimization you are speaking of, in a way that is not fragile to moving

Re: Share .cpp and .h along projects

source files between DLLs, changing optimization settings, etc.

Provided that "lock" and "unlock" trigger memory barriers, the code you show will be correct on any version of the C++ compiler (even non-Microsoft compilers), using any optimization settings, if you would only say "volatile int x".

In fact, if your "x" is in an anonymous namespace, which would be the recommended practice, and you do not take its address, then the compiler aliasing analysis can determine that it will not be touched by the "opaque" DLL, and reorder or remove the access to x once again.

You ***MUST*** use "volatile" in this case to guarantee correct behavior, and when you do, whether the lock and unlock functions are in a DLL is no longer important.

The DCLP issue is similar, in that placing the ctor of the created object in an opaque DLL prevents the compiler from updating the local static pointer before the ctor has finished.

No. Unless you have a memory barrier, other threads could see the change to the pointer before the change to the members of the object are visible. Your games with DLLs will not save you. "volatile" will save you in VC2005, and "volatile" combined with InterlockedExchangePointer to update the pointer is the correct, portable way of doing so.