Ensuring protected access to shared data

November 2014

1 lockstrap

A few macros to keep data and it's lock together, for C++11 and later.

In multithreaded programming, locks (mostly mutexes) are used a lot. Actually, a lot more than they should be, but that is another topic.

The problem with locks is that there is no way to associate a lock with some data. Let's say you have class like this:

```
class User {
    int a;
    float b;
    std::string c;
    std::vector<long> x;

    std::mutex m;
    // ...
};
```

Quick question: Which data does 'm' protect?

You can't tell. It might be all, but then it might not. It would be even worse if there were two mutexes in that class. You have to actually look at the code to figure it out. And code can be a mess. You can resort to comments (good luck with that) or naming conventions (which quickly get out of hand).

But even if your comments are good or naming conventions are feasable, there is nothing preventing bad usage - protecting data when it shouldn't be and vice versa.

Well, these "lock-strap" macros provide a way to fix that. I know, I wish they were not macros, but I'm pretty certain that with C++11 through C++23 there is no way to avoid macros *and* keep a nice syntax for the users.

1.1 Usage

Using the same example as before:

```
#include "lockstrap.h"
class User {
```

```
    class Data {
        int a;
        float b;
        std::string c;
        LOCKSTRAP(Data, std::mutex, a,b,c);
    } d;
    std::vector<long> x;
    // ...
};
```

So, we introduced a class to hold the data which is protected by a mutex (which is declared in the 'Data' class by the "LOCKSTRAP" macro). The vector 'x' is, now obviously, not protected by the mutex.

Since 'Data' is a class, obviously you can't access 'a', 'b' and 'c' from the outside. The 'LOCKSTRAP' macro defines two helper member functions for such purposes - the 'access' and the template 'with'.

This is how you would use them in some User member function(s):

```
void User::f()
{
    auto al = d.access();
    // at this point, mutex is locked, you can access and do..
    // whatever.
    al.a = 3;
    al.b = al.a / 2;
    // mutex will be unlocked when 'al' goes out of scope here
}

void User::g()
{
    // In C++14, 'auto' can replace 'User::locker'
    d.with([](User::locker l) {
        // at this point, mutex is locked, you can access and do
            ..
        // whatever.
        l.a = 33;
        l.c.append(std::to_string(l.b));
        // mutex will be unlocked when 'l' goes out of scope
            here
    });
    // with() accepts any callable object, even a function
        pointer.
}
```

You can combine both 'access' and 'with' in the same function, but that would be strange.

1.2 The simple implementation

Simple implementation has very similar usage. Include "lockstrap_simple.h" instead of "lockstrap.h" and use "LCKSTRAPSIMP" instead of "LOCKSTRAP" macro, and when

accessing data always use function call syntax. Here's the full example with simple implementation:

```
#include "lockstrap_simple.h"
class User {
    class Data {
        int a;
        float b;
        std::string c;
        LCKSTRAPSIMP(Data, std::mutex, a,b,c);
    } d;
    std::vector<long> x;
    // ...
};

void User::f()
{
    auto al = d.access();
    al.a() = 3;
    al.b() = al.a() / 2;
}

void User::g()
{
    d.with([](User::locker l) { l.c().append(std::to_string(l.b
        ())); });
}
```

You can use the "simple" implementation for some classes and the regular for others, but doing that in the same file would be a little confusing to the reader.

1.3 Why use simple at all?

Well, you may prefer this syntax with all those (), as it hints that this is not your regular access.

Even if you don't, It compiles faster. How faster depends, on how you use it. But, in basic tests when code did little else but access this data, it was about 15% faster. With more code non-lock related, the relative speedup will be smaller, but, OTOH, if you have a lot of code with locking than it might be a significant absolute speedup.

In theory, the "regular" implementation may generate worse code, because it declares a "shadow" reference for each protected data member. In all tests, especially with optimizations on, the generated code is actually the same, as everything is inlined.

Also, for most compilers, simple will give somewhat nicer errors on incorrect usage.

1.4 Remarks

The lock doesn't have to be a mutex. It can be anything that implements the "Lockable" concept, that is, has a "lock()" and "unlock()" member functions.

Since you have to "mention" every data member, you may make a mistake:

- If you omit a member, compiler will give you a "class XXX::locker does not have a member . . . " error which is a good hint that you have to add it.
- If you give bad data member name, compiler will give an error like "XXX::bad_name doesn't exist", which is also a good hint

The initial implementation can handle up to 9 data members. It is easy to add more, look at the comments in the headers.

Obviously, this was not designed to handle static data (though it will work, to an extent) or member functions - which will give strange compiler errors - well, you know that member functions can't be protected by a mutex, right?

1.5 Implementation

Implementation is nothing special:

- The macro generates a lock data member in class itself
- Then it generates a nested class named "locker" which will be used to access the data
- For simple implementation, the "locker" has a reference to the "real" object and has a bunch of member functions with the same names as the data members of the "real" class
- For regular implementation, the "locker" has a reference for each data member of the real class, with exactly the same name
- In any case, "locker" will lock the lock on construction and unlock on destruction (yup, RAII)
- The macro generates an "access" member function which will return a "locker" object
- The macro generates a "with" member template function which will accept a templatized callable parameter and create a "locker" object which it will pass to that parameter

1.6 Discussion

1.6.1 Why can't this be done with templates?

Well, you may do something like a "smart pointer" if you make the data "public" in the "real" class, but that defies the purpose, as the data is now public an anyone can use it without the lock.

If you keep the data private, then you may use something like a tuple, but that doesn't generate the symbol names, so you would use different (and rather ugly) syntax to access the data. There are other tricks to be done here, but all with the same "tuple" problem.

C++11 makes the implementation and usage a lot easier, with variable arguments macros and decltype. Actually, with C++14, you can use 'auto' instead of 'decltype(ME::x)' when declaring the "forwarding functions" in the "simple" implementation.

1.6.2 C++03

One could do similar stuff in C++03, but would have to re-declare the types of all data (and call the macro "version" with the right number of data):

```
class Data {
    int a;
    float b;
    std::string c;
    LCKSTRAPSIMP3(Data, std::mutex, int, a, float, b, std::
        string, c);
} d;
```

or go with something like:

```
DECL_LOCKSTRAP_CLASS(User, std::mutex)
DECL_LOCKSTRAP_MEMBER(int, a);
DECL_LOCKSTRAP_MEMBER(float, b);
END_LOCKSTRAP_CLASS()
```

Herb Sutter has a C++03 design which is similar to this in an article on DrDobbs Journal. Last known URL:

http://www.drdobbs.com/windows/associate-mutexes-with-data-to-prevent-r/224701827

Except the rather ugly macro syntax, his design actually exposes lock() and unlock(), which makes it error prone (you may access the data without locking). He does check (that lock is locked/held) with an assert, but, I believe that lockstrap design is better, as asserts aren't there in release configuration, and mutlithreading bugs are notorious for manifesting (only) in the field (and you ship release configuration to the field). Also, even in the debug build, this assert is not enough. That is, you may assert that lock is locked, but, by the time you access the data, lock might get unlocked (from another thread, of course).

1.6.3 Is parameter of the with() to be passed by value or rvalue?

Well, it is passed as a rvalue by design. Whether you declare it as value parameter, like:

```
    d.with([](User::locker l) {
```

or rvalue parameter, like:

```
    d.with([](User::locker &&l) {
```

does not matter much. In theory it might, but in practice, especially if optimizations are turned on, this produces the same machine/binary code.

2 Regular/full implementation

This should be saved to `lockstrap.h`.

```
#if !defined INC_LOCKSTRAP
#define      INC_LOCKSTRAP


/* These internal macros do "step macro development". Don't use them
   outside this header. But, if you ever make a class with more than
   elements (thus steps) present here , just do some copy pasting and
   add more steps...
 */
```

```
#define LCKSTRPDCL1(ME, x) decltype(ME::x) &x
#define LCKSTRPDCL2(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL1(ME,
    __VA_ARGS__)
#define LCKSTRPDCL3(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL2(ME,
    __VA_ARGS__)
#define LCKSTRPDCL4(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL3(ME,
    __VA_ARGS__)
#define LCKSTRPDCL5(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL4(ME,
    __VA_ARGS__)
#define LCKSTRPDCL6(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL5(ME,
    __VA_ARGS__)
#define LCKSTRPDCL7(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL6(ME,
    __VA_ARGS__)
#define LCKSTRPDCL8(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL7(ME,
    __VA_ARGS__)
#define LCKSTRPDCL9(ME, x, ...) decltype(ME::x) &x; LCKSTRPDCL8(ME,
    __VA_ARGS__)

/* Ditto... */
#define LCKSTRPINIT1(ME, x), x(me.x)
#define LCKSTRPINIT2(ME, x, ...), x(me.x) LCKSTRPINIT1(ME, __VA_ARGS__)
#define LCKSTRPINIT3(ME, x, ...), x(me.x) LCKSTRPINIT2(ME, __VA_ARGS__)
#define LCKSTRPINIT4(ME, x, ...), x(me.x) LCKSTRPINIT3(ME, __VA_ARGS__)
#define LCKSTRPINIT5(ME, x, ...), x(me.x) LCKSTRPINIT4(ME, __VA_ARGS__)
#define LCKSTRPINIT6(ME, x, ...), x(me.x) LCKSTRPINIT5(ME, __VA_ARGS__)
#define LCKSTRPINIT7(ME, x, ...), x(me.x) LCKSTRPINIT6(ME, __VA_ARGS__)
#define LCKSTRPINIT8(ME, x, ...), x(me.x) LCKSTRPINIT7(ME, __VA_ARGS__)
#define LCKSTRPINIT9(ME, x, ...), x(me.x) LCKSTRPINIT8(ME, __VA_ARGS__)

// The generic macro to help with "step macro development". If you
// ever make a class with more than elements present here, just add
// parameters before the "NAME" parameter
#define GET_MACRO(_1,_2,_3,_4,_5,_6,_7,_8,_9,NAME,...) NAME

/** Define a safe lock "strap" (or band, binder...) over data of a
    class. It will add a "lock object" of the type of your choice to
    your class and enable you to access the data only under that lock.

    This is the macro to be used outside this header, like this:

    class MyClass {
        int my_int;
        float my_float;
        LOCKSTRAP(MyClass, std::mutex, my_int, my_float);
    };

    Then, use the class like this:

    MyClass x;
    auto xl = x.access();
    xl.my_int = 5;
    std::cout << xl.my_int << std::endl;
    x.with([](MyClass::locker l) { l.my_int = 6 });
    // or, in C++14:
    x.with([](auto l) { l.my_int = 6 });
```

```
    Remember to keep the GET_MACRO "calls" up-to-date with the maximum
    number of steps supported.

    @param ME The name of the class we are adding a "lock strap" to
    @param LOCK The type of the lock (implements the "Lockable"
    concept - i.e., has a "lock()" and "unlock()" member functions.)
 */
#define LOCKSTRAP(ME, LOCK, ...)
            \
  private: LOCK d_locker;
              \
public: class locker {
            \
    ME &d_me;
                \
public:
            \
 GET_MACRO(__VA_ARGS__, LCKSTRPDCL9,LCKSTRPDCL8,LCKSTRPDCL7,LCKSTRPDCL6
      ,LCKSTRPDCL5,LCKSTRPDCL4,LCKSTRPDCL3,LCKSTRPDCL2,LCKSTRPDCL1)(ME,
       __VA_ARGS__); \
 locker(ME &me) : d_me(me) GET_MACRO(__VA_ARGS__, LCKSTRPINIT9,
      LCKSTRPINIT8,LCKSTRPINIT7,LCKSTRPINIT6,LCKSTRPINIT5,LCKSTRPINIT4,
      LCKSTRPINIT3,LCKSTRPINIT2,LCKSTRPINIT1)(ME, __VA_ARGS__) \
  { me.d_locker.lock(); }
              \
 ~locker() { d_me.d_locker.unlock(); }
               \
};
            \
  locker access() { return locker(*this); }
              \
  template <typename F> void with(F f) { f(locker(*this)); }


#endif // !defined INC_LOCKSTRAP
```

3 Simple implementation

This should be saved to `lockstrap_simple.h`.

```
#if !defined INC_LCKSTRAP_SIMPLE
#define      INC_LCKSTRAP_SIMPLE

/* These internal macros do "step macro development". Don't use them
   outside this header.  But, if you ever make a class with more than
   elements (thus steps) present here , just do some copy pasting and
   add more steps...
 */
#define LCKSTRAPSIMP1(ME, x) decltype(ME::x) &x() {return d_me.x;}
#define LCKSTRAPSIMP2(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP1(ME, __VA_ARGS__)
#define LCKSTRAPSIMP3(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP2(ME, __VA_ARGS__)
#define LCKSTRAPSIMP4(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP3(ME, __VA_ARGS__)
```

```
#define LCKSTRAPSIMP5(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP4(ME, __VA_ARGS__)
#define LCKSTRAPSIMP6(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP5(ME, __VA_ARGS__)
#define LCKSTRAPSIMP7(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP6(ME, __VA_ARGS__)
#define LCKSTRAPSIMP8(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP7(ME, __VA_ARGS__)
#define LCKSTRAPSIMP9(ME, x, ...) decltype(ME::x) &x() {return d_me.x;}
     LCKSTRAPSIMP8(ME, __VA_ARGS__)

// The generic macro to help with "step macro development". If you
// ever make a class with more than elements present here, just add
// parameters before the "NAME" parameter
#define GET_MACRO(_1,_2,_3,_4,_5,_6,_7,_8,_9,NAME,...) NAME

/** Define a safe lock "strap" (or band, binder...) over data of a
    class. It will add a "lock object" of the type of your choice to
    your class and enable you to access the data only under that lock.

    This is the macro to be used outside this header, like this:

    class MyClass {
        int my_int;
        float my_float;
        LCKSTRAPSIMP(MyClass, std::mutex, my_int, my_float);
    };

    Then, use the class like this:

    MyClass x;
    auto xl = x.access();
    xl.my_int() = 5;
    std::cout << xl.my_int() << std::endl;
    x.with([](MyClass::locker l) { l.my_int() = 6 });
    // or, in C++14:
    x.with([](auto l) { l.my_int() = 6 });

    Remember to keep the GET_MACRO "call" up-to-date with the maximum
    number of steps supported.

    @param ME The name of the class we are adding a "lock strap" to
    @param LOCK The type of the lock (implements the "Lockable"
    concept - i.e., has a "lock()" and "unlock()" member functions.)
 */
#define LCKSTRAPSIMP(ME, LOCK, ...)
            \
  private: LOCK d_locker;
              \
public: class locker {
            \
    ME &d_me;
                \
public: locker(ME &me) : d_me(me) { me.d_locker.lock(); }
            \
    ~locker() { d_me.d_locker.unlock(); }
```

```
                \
    GET_MACRO(__VA_ARGS__, LCKSTRAPSIMP9,LCKSTRAPSIMP8,LCKSTRAPSIMP7,
        LCKSTRAPSIMP6,LCKSTRAPSIMP5,LCKSTRAPSIMP4,LCKSTRAPSIMP3,
        LCKSTRAPSIMP2,LCKSTRAPSIMP1)(ME, __VA_ARGS__) \
};
            \
  locker access() { return locker(*this); }
              \
  template <typename F> void with(F f) { f(locker(*this)); }


#endif //!defined INC_LCKSTRAP_SIMPLE
```