Sharing data between threads

Problems start when you want to modify shared variables. Having readonly variables shared between threads should not create problem.

The concept of invariants

One concept that's widely used to help programmers reason about their code is *invariants* — statements that are always true about a particular data structure, such as "this variable contains the number of items in the list". These invariants are often broken during an update, especially if the data structure is of any complexity or the update requires modification of more than one value.

Race conditions

Race conditions : anything where the outcome depends on the relative ordering of execution of operations on two or more threads; the threads race to perform their respective operations.

Problematic race conditions typically occur where completing an operation requires modification of two or more distinct pieces of data. Because the operation must access two separate pieces of data, these must be modified in separate instructions, and another thread could potentially access the data structure when only one of them has been completed. Race conditions can often be hard to find and hard to duplicate because the window of opportunity is small. Because race conditions are generally timing-sensitive, they can often disappear entirely when the application is run under the debugger, because the debugger affects the timing of the program, even if only slightly.

Ways to avoid race conditions :

- Wrap your data structure with a protection mechanism to ensure that only the thread performing a modification can see the intermediate states where the invariants are broken.
- Another option is to modify the design of your data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants. This is generally referred to as *lock-free programming* and is difficult to get right. (Used for realtime programming.)
- Another way of dealing with race conditions is to handle the updates to the data structure as a transaction, just as updates to a database are done within a transaction. This is termed *software transactional memory* (STM), and it's an active research area.

Protect data with Mutex

Definition

Mutex : MUTual EXclusion

Before accessing a shared data structure, you lock the mutex associated with that data, and when you've finished accessing the data structure, you unlock the mutex. The Thread Library then ensures that once one thread has locked a specific mutex, all other threads that try to lock the same mutex have to wait until the thread that successfully locked the mutex unlocks it. This ensures that all threads see a self-consistent view of the shared data, without any broken invariants.

Mutexes are the most general of the data-protection mechanisms available in C++, but they're not a silver bullet :

- The wait they imply is not adapted to realtime
- It's important to structure your code to protect the right data
- You must avoid race conditions inherent in your interfaces
- Mutexes also come with their own problems in the form of a deadlock
- Protecting either too much or too little data will be a problem

Mutex Example

In C++, you create a mutex by constructing an instance of `std::mutex`, lock it with a call to the lock() member function, and unlock it with a call to the unlock() member function. But it isn't recommended practice to call the member functions directly, because this means that you have to remember to call unlock() on every code path out of a function, including those due to exceptions. Instead, the Standard C++ Library provides the `std::lock_guard` class template, which implements that RAII idiom for a mutex; it locks the supplied mutex on construction and unlocks it on destruction, ensuring a locked mutex is always correctly unlocked.

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> someList;
std::mutex someMutex;

void addToList(int newValue) {
    std::lock_guard<std::mutex> guard(someMutex);       // A
    someList.push_back(newValue);
}

bool listContains(int valueToFind) {
    std::lock_guard<std::mutex> guard(someMutex);       // A
    return std::find(someList.begin(), someList.end(), valueToFind) != someList.end();
}
```

C++17 has a new feature called class template argument deduction, which means that for simple class templates like `std::lock_guard`, the template argument list can often be omitted. A and B can be reduced to:

```
std::lock_guard guard(someMutex);
```

Usually, you would group the mutex and the protected data together in a class rather than use global variables. In this case, the *addToList* and *listContains* functions would become member functions of the class, and the mutex and protected data would both become private members of the class, making it much easier to identify which code has access to the data and thus which code needs to lock the mutex. If all the member functions of the class lock the mutex before accessing any other data members and unlock it when done, the data is nicely protected from all comers.

EXCEPT in one case : any code that has access to that pointer or reference can now access (and potentially modify) the protected data without locking the mutex. Protecting data with a mutex therefore requires careful interface design.

It's also important to check that they don't pass these pointers or references in to functions they call that aren't under your control. This is just as dangerous: those functions might store the pointer or reference in a place where it can later be used without the protection of the mutex.

```
class SomeData
{
    int a;
    std::string b;
public:
    void doSomething();
};

class DataWrapper
{
private:
    SomeData data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> l(m);
        func(data);
    }
};

SomeData* unprotected;

void maliciousFunction(SomeData& protectedData)
{
    unprotected=&protectedData;
}

DataWrapper x;
void foo()
{
    x.process_data(maliciousFunction);
    unprotected->doSomething();        // Unprotected access to protected data
}
```

A rule to help you : don't pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions.

Racing conditions in interfaces

Using mutex doesn't mean you're protected from race conditions; you still have to ensure that the appropriate data is protected. Consider a doubly linked list, where you want to remove a node. In order for a thread to safely delete a node, you need to ensure that you're preventing concurrent accesses to three nodes: the node being deleted and the nodes on either side.

If you protected access to the pointers of each node individually, you'd be no better off than with code that used no mutexes, because the race condition could still happen — it's not the individual nodes that need protecting for the individual steps but the whole data structure, for the whole delete operation.

This problem happens as a consequence of the design of the interface, so the solution is to change the interface.

Deadlock

Imagine you have threads arguing over locks on mutexes: each of a pair of threads needs to lock both of a pair of mutexes to perform some operation, and each thread has one mutex and is waiting for the other. Neither thread can proceed, because each is waiting for the other to release its mutex. This scenario is called *deadlock*.

Avoiding pair deadlock

The common advice for avoiding deadlock is to always lock the two mutexes in the same order: if you always lock mutex A before mutex B, then you'll never deadlock.

Sometimes this is straightforward, because the mutexes are serving different purposes, but other times it's not so simple, such as when the mutexes are each protecting a separate instance of the same class. Thankfully, the C++ Standard Library has a cure for this in the form of `std::lock` — a function that can lock two or more mutexes at once without risk of deadlock.

```
class BigObject;
void swap(BigObject& lhs,BigObject& rhs);

class X {
private:
    BigObject someDetail;
    std::mutex m;
public:
    X(BigObject const& sd) : someDetail(sd) {}

    friend void swap(X& lhs, X& rhs) {
        if (&lhs==&rhs)                 // Ensure they are different instances
            return;
        std::lock(lhs.m,rhs.m);         // Lock the two mutex

        std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
        std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
        swap(lhs.someDetail, rhs.someDetail);
    }
};
```

The `std::adopt_lock` parameter is supplied in addition to the mutex to indicate to the `std::lock_guard` objects that the mutexes are already locked, and they should adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor.

This ensures that the mutexes are *correctly* unlocked on function exit in the general case where the protected operation *might* throw an exception; it also allows for a simple return. Also, it's worth noting that locking either lhs.m or rhs.m inside the call to `std::lock` can throw an exception; in this case, the exception is propagated out of `std::lock`. If `std::lock` has successfully acquired a lock on one mutex and an exception is thrown when it tries to acquire a lock on the other mutex, this first lock is released automatically: `std::lock` provides all-or-nothing semantics with regard to locking the supplied mutexes.

Other advices to avoid deadlocks

You can create deadlock with two threads and no locks by having each thread call join() on the `std::thread` object for the other. The guidelines for avoiding deadlock all boil down to one idea: don't wait for another thread if there's a chance it's waiting for you.

Design a thread-safe stack

In this paragraph we will go through the process of making a classic data structure thread-safe.

Why std::stack is not thread safe

Let's observe the `std::stack` interface:

```
template<typename T,typename Container=std::deque<T> >
class Stack
{
public:
    explicit Stack(const Container&);
    explicit Stack(Container&& = Container());
    template <class Alloc> explicit Stack(const Alloc&);
    template <class Alloc> Stack(const Container&, const Alloc&);
    template <class Alloc> Stack(Container&&, const Alloc&);
    template <class Alloc> Stack(Stack&&, const Alloc&);
    bool empty() const;
    size_t size() const;
    T& top();
    T const& top() const;
    void push(T const&);
    void push(T&&);
    void pop();
    void swap(Stack&&);
    template <class... Args> void emplace(Args&&... args);
};
```

The problem here is that the results of empty() and size() can't be relied on. Although they might be correct at the time of the call, once they've returned, other threads are free to access the stack and might push() new elements onto or pop() the existing ones off the stack before the thread that called empty() or size() could use that information.

In particular, if the stack instance is not shared, it's safe to check for empty() and then call top() to access the top element if the stack is not empty, as follows:

```
stack<int> s;
if(!s.empty()) {
    int const value=s.top();
    s.pop();
    doSomething(value);
}
```

With a shared stack object, this call sequence is no longer safe, because there might be a call to pop() from another thread that removes the last element in between the call to empty() B and the call to top() C. This is therefore a classic race condition, and the use of a mutex internally to protect the stack contents doesn't prevent it; it's a consequence of the interface.

Here is an other problem/ If you run the code above with two threads A and B on the same stack, you can have the following order of execution:

```
A: if(!s.empty())
B: if(!s.empty())
A:     int const value=s.top();
B:     int const value=s.top();
A:     s.pop();
A:     doSomething(value);
B:     s.pop();
B:     doSomething(value);
```

One of the value of the stack is thus discarded without having been read, and the other is processed twice.

To solve those problems we can consider differen.

A solution

We will implement a solution with two options :

- Passing by reference in pop() the variable in which you want to store the result. This is not always possible thought, given the type of the stack content.
- Return a pointer to the poped variable. For any interface that uses this option, `std::shared_ptr` would be a good choice of pointer type; not only does it avoid memory leaks, because the object is destroyed once the last pointer is destroyed, but the library is in full *control* of the memory allocation scheme.

We will use mutex to *avoid* racing conditions.

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>

struct EmptyStackException: std::exception
{
    const char* what() const throw();
};

template<typename T>
class ThreadsafeStack {
private:
    std::stack<T> data;
    mutable std::mutex m;

public:
    ThreadsafeStack(){}

    ThreadsafeStack(const ThreadsafeStack& other) {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }

    ThreadsafeStack& operator=(const ThreadsafeStack&) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw EmptyStackException();
        std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
        data.pop();
        return res;
    }

    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw EmptyStackException();
        value=data.top();
        data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

This stack implementation is copyable—the copy constructor locks the mutex in the source object and then copies the internal stack