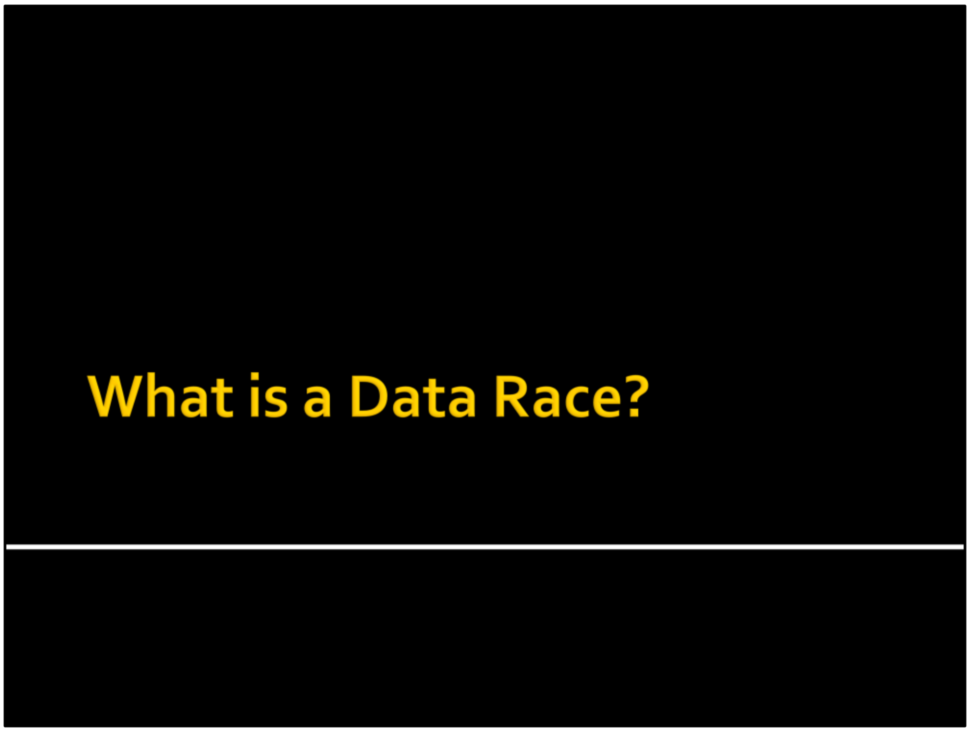
Bartosz Milewski

Ownership Systems against Data Races

The biggest two problems in multi-threaded programming are **races** and **deadlocks**. Races reached new levels with the introduction of relaxed memory processors.

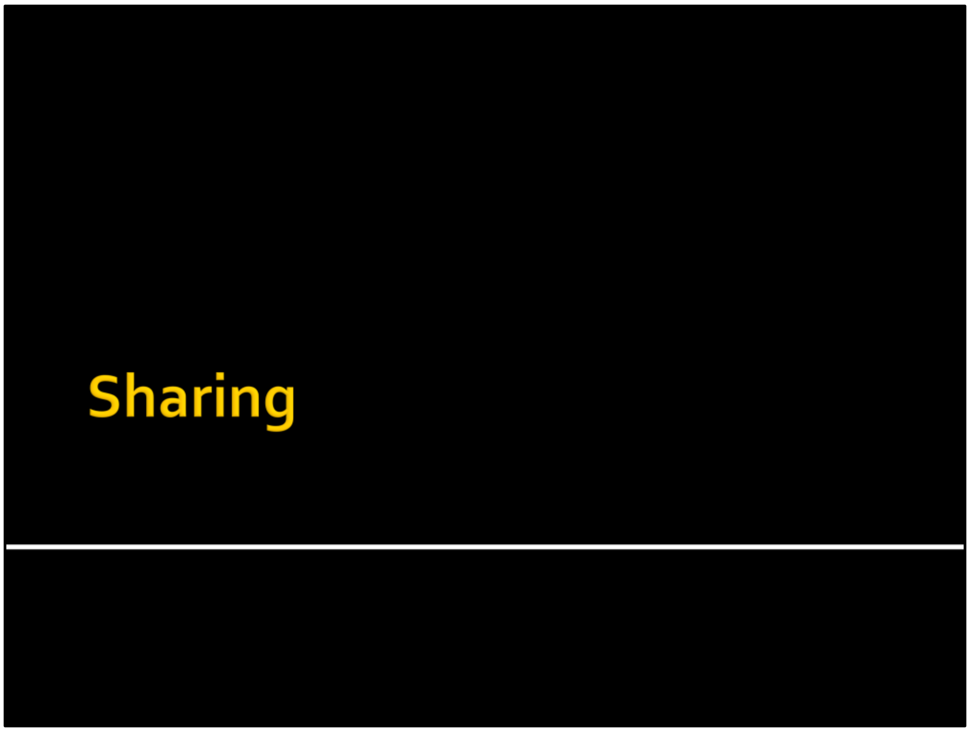
It turns out that races can be eliminated without sacrificing much in terms of performance or expressive power.

It is based on published peer-reviewed work on type systems that guarantee race freedom. We can't expect C++ to extend its type system, but we can, with a little discipline, follow a methodology based on it.



What is a Data Race?

What is data race? Multiple threads accessing a shared location, at least one of them a writer, with no synchronization. All three conditions must be true for a data race:
-Sharing
-Mutation
-Lack of synchronization



Sharing

Reduce and control sharing.

Processes vs. threads. Erlang processes—no sharing, guaranteed by the language (not the OS). Advantage: no data races, location transparency (don't care if processes run within the same OS process, or on multiple machines communicating over a network).

Threads offer shared (mutable) memory. Great for speed but marred by data races.

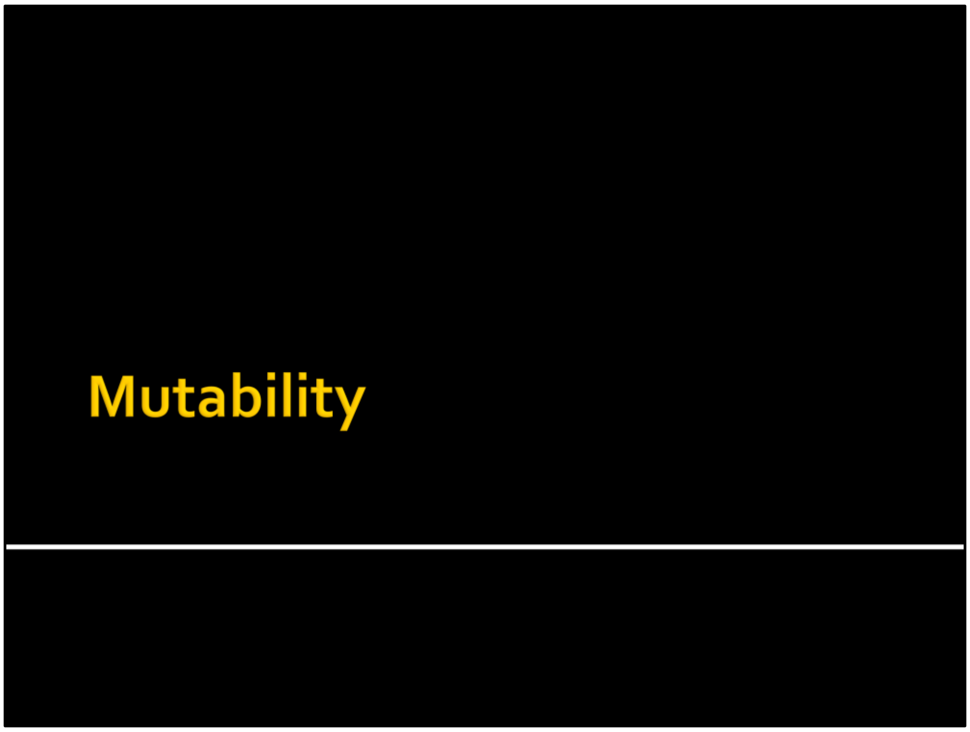
Greatest problem: **accidental sharing**.

-Avoid globals!
-Don't share local (stack) variables—they might disappear
-What about heap objects (created using operator new)? They are usually passed by reference (although C++ allows value semantics through copy constructor/ assignment overloads).

Convention: Clearly distinguish shared from non-shared—we'll talk about annotations that help doing that.

In some languages (D), there is a type modifier "shared" checked by the compiler. In C++ use comments or macros to mark shared objects. Shared may be subdivided into finer categories.

How does data become shared (other than through globals)? A program starts as single-threaded. Access to shared data is passed during thread creation and through



Mutability

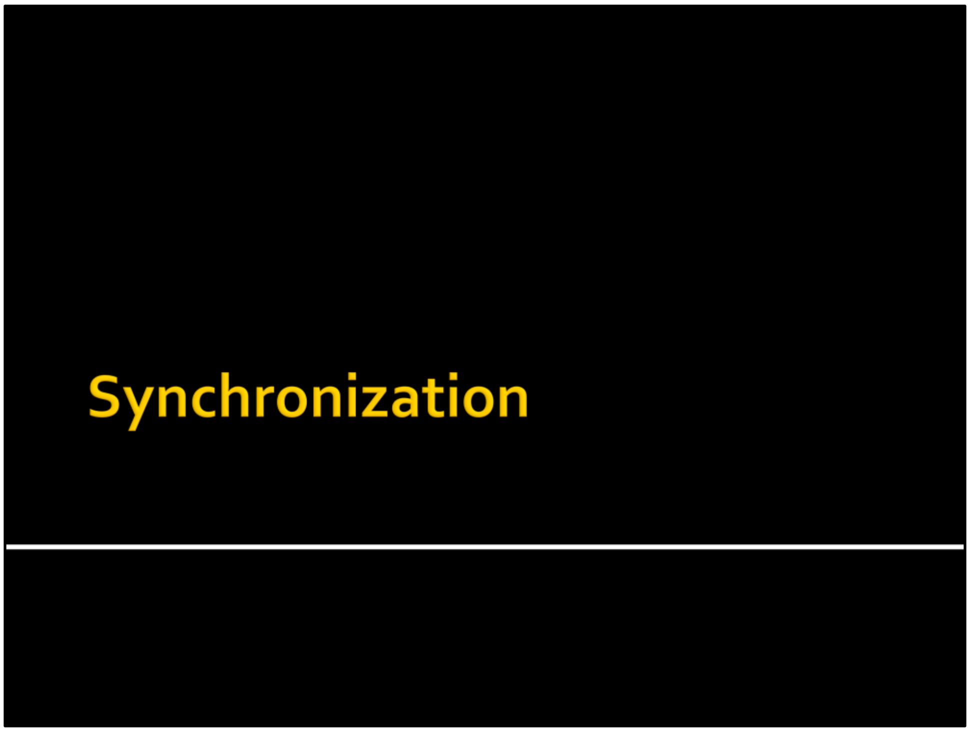
In some languages (pure functional) everything is immutable. No danger of data races. But even Haskell relaxes this rule (MVars).

In C++ there is no “immutable”, only “const”. The problem: the same object may be observed through const reference as well as a non-cont one. Also “const” is **shallow**. (D offers both deep immutable and deep const.)

So again, we are left with a convention and comments.

Make sure your immutable is deep—everything an immutable object has access to must be immutable (or a monitor, see later).

Beware of **publication** problems. Immutable objects may be, and often are, mutated during creation. Even if nobody modifies them later, other threads might see them semi-constructed. To avoid this, construct them before spawning threads that share them. Or use atomic pointers for sharing.



Synchronization

Access to any mutable shared data must be synchronized.

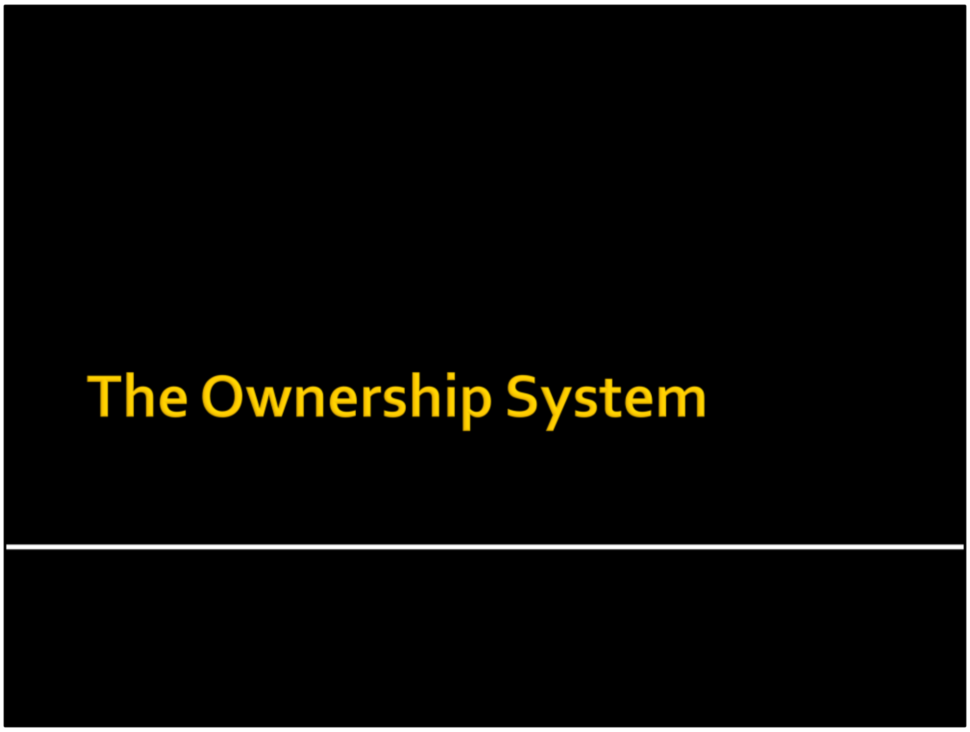
In Java, use *synchronized* methods and sections.

In C++ use mutexes and locks.

Important: Both write access AND **read access** must be protected by the same lock. What's the worst thing that may happen if you don't protect reads? Getting stale value? Not only. The compiler/processor may rearrange your reads changing the **logic** of your program. Cautionary tale: Double-Checked Locking Pattern.

If you don't use locks you are doing **lock-free programming**—you have been warned!

If you like to live on the edge and insist on lock-free programming, use "volatile" in Java and strong atomics in C++. Don't even think of using weak atomics!



The Ownership System

It’s possible to statically eliminate data races (compiler-checked). The key is the type system based on ownership.

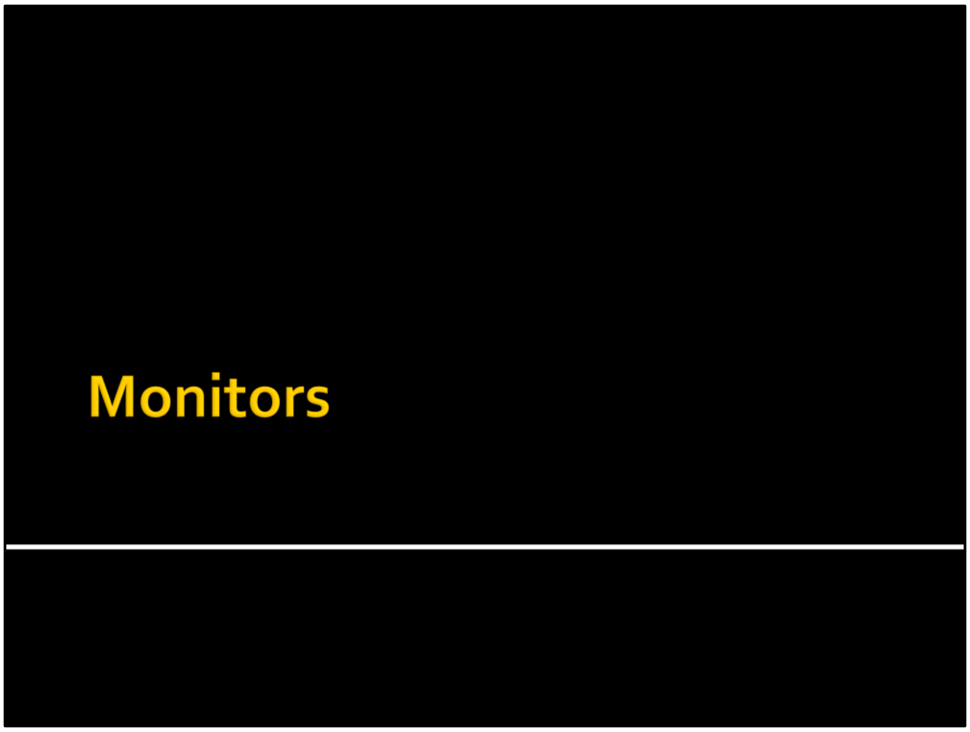
Every piece of data has one and only one **top owner**. To access any object you must lock its top owner. To find the top owner follow the ownership chain. In the ownership-based type system the compiler is able to verify this at compile time.

The owner is either a lock or another object. Owners are individual instances. There is also a symbolic top owner “thread” for thread-local objects (non-shared).

The top owner is for life—assigned during construction. Therefore the ownership relation generates trees. The top of every tree is either “thread” or a lock. (”thread” is considered locked at all times.)

We don’t have the luxury of an ownership type system, so we have to use annotations and discipline.

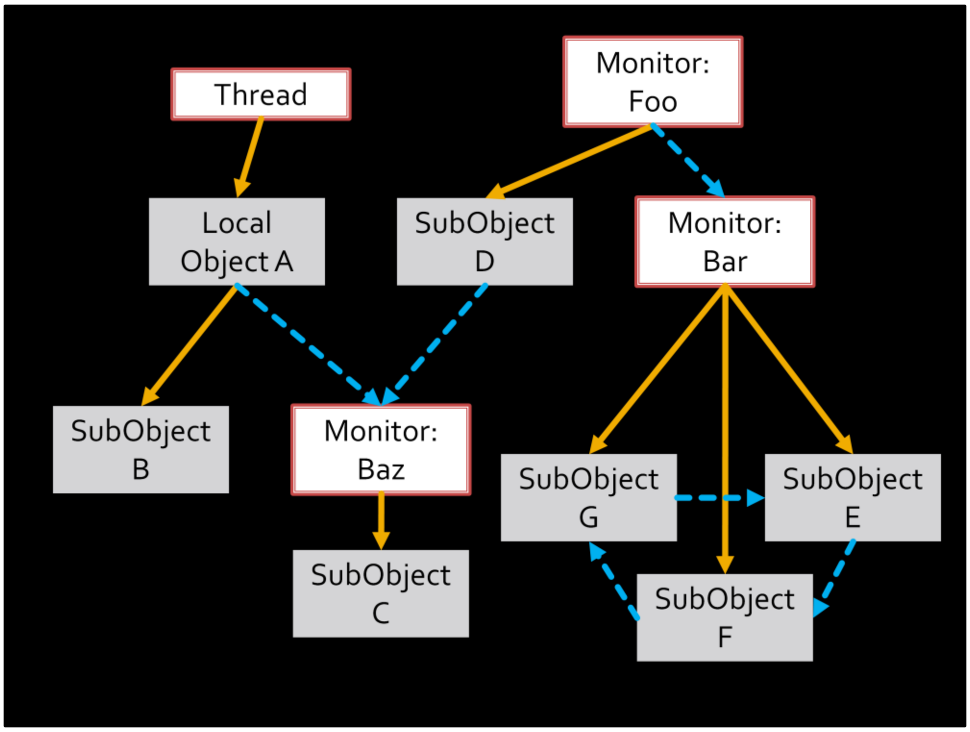
Big advantage of type system based approach: all checks are **local**!



Monitors

In an OO language, ownership is best expressed by containment. An object owns its subobjects (see however exceptions to the rule).

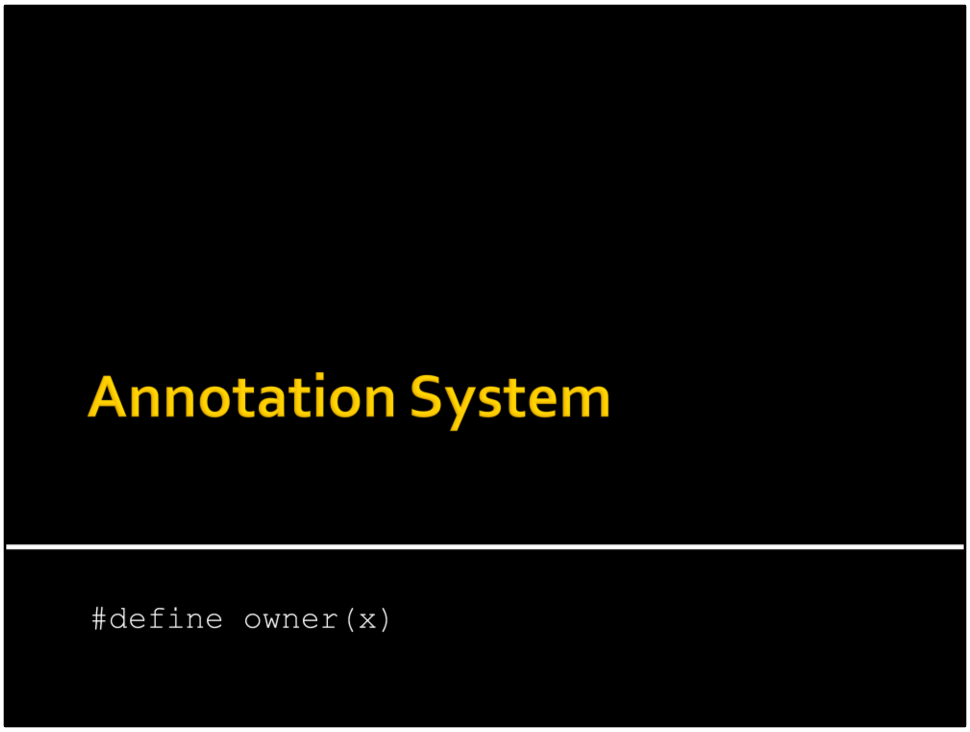
A monitor is an object that contains its own lock (and is potentially shared). A monitor is self-owned. Its subobjects are owned by the monitor. Access to any part of the tree may only happen if the monitor's lock is taken.



Yellow lines: owns.
Blue dashed lines: contains/has access to (but not owns).

A monitor may *not* contain (or have access to) any thread-owned objects.

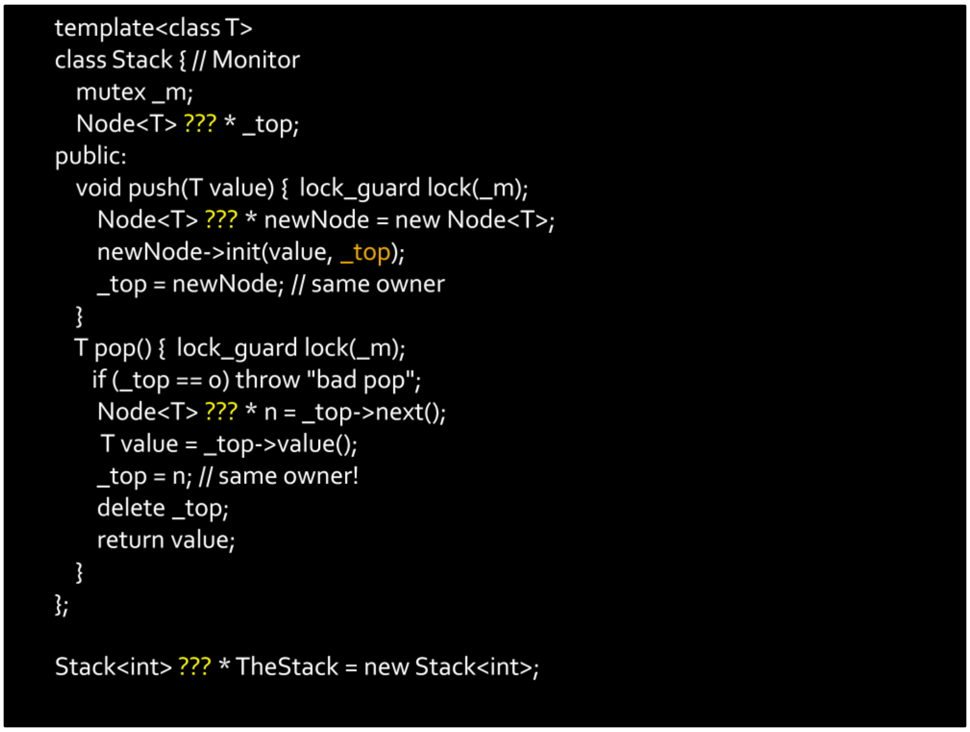
Any object may contain (or have access to) other monitors. Monitors may be safely shared. Embedded monitors are again self-owned and create their own trees.



Annotation System

```
#define owner(x)
```

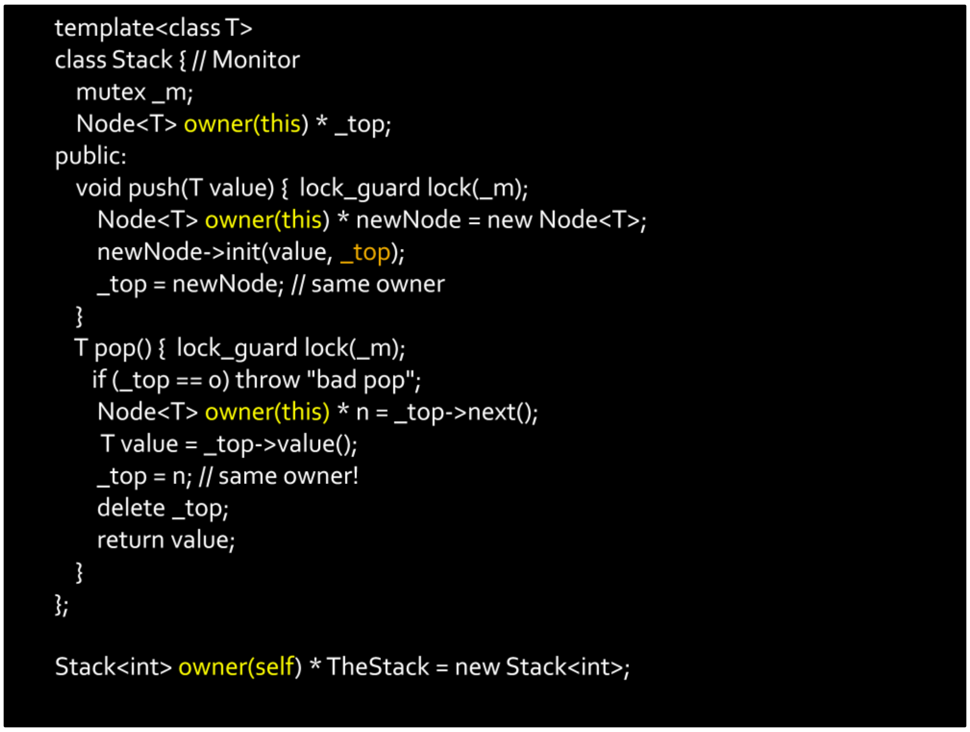
This macro expands to nothing. It helps the programmer keep track of ownership.



```
template<class T>
class Stack { // Monitor
  mutex _m;
  Node<T> ??? * _top;
public:
  void push(T value) {  lock_guard lock(_m);
    Node<T> ??? * newNode = new Node<T>;
    newNode->init(value, _top);
    _top = newNode; // same owner
  }
  T pop() {  lock_guard lock(_m);
    if (_top == 0) throw "bad pop";
    Node<T> ??? * n = _top->next();
    T value = _top->value();
    _top = n; // same owner!
    delete _top;
    return value;
  }
};

Stack<int> ??? * TheStack = new Stack<int>;
```

Stack is a monitor: it contains its own mutex and all methods lock that mutex.
_top is owned by the stack, owner(this). When the class is instantiated, this becomes the instance of the class.
push: new node created with owner(this)
  _top is passed to Node::init, make sure init method of a stack-owned node accepts stack-owned node
    assignment _top = newNode: ownership must match.
pop: assignment _top = _top.next(), make sure _top.next() returns a node whose owner is compatible with the owner of this. Compatible means “from the same ownership tree.” In this case _top is owned by stack. next() returns an object owned by _top (see next slide), so they are compatible.

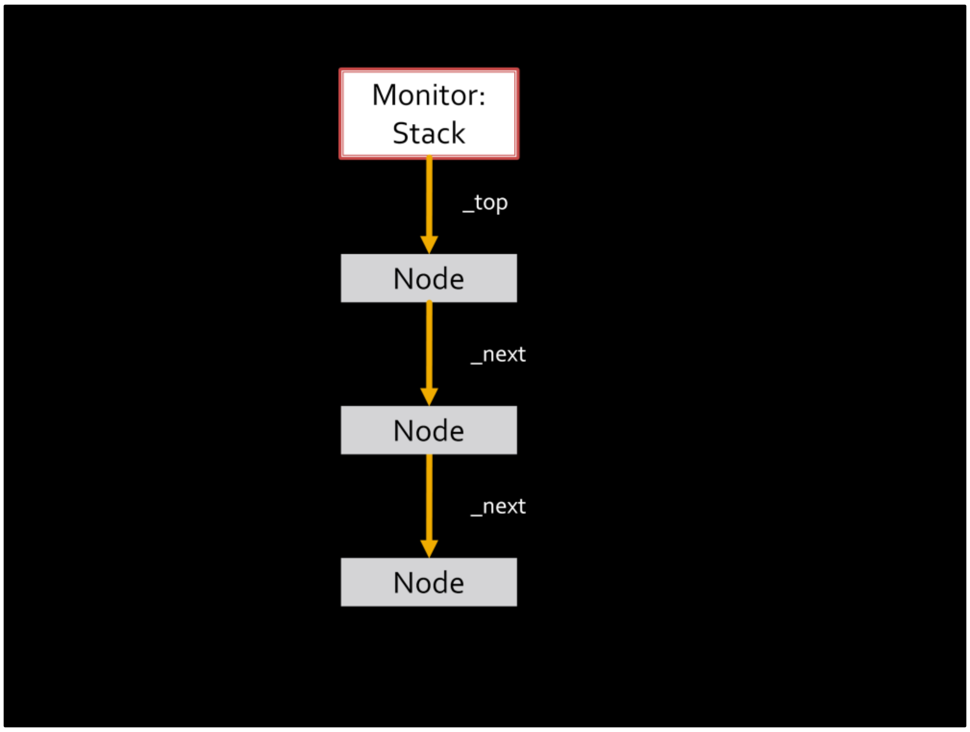
TheStack, being a monitor, is self-owned.



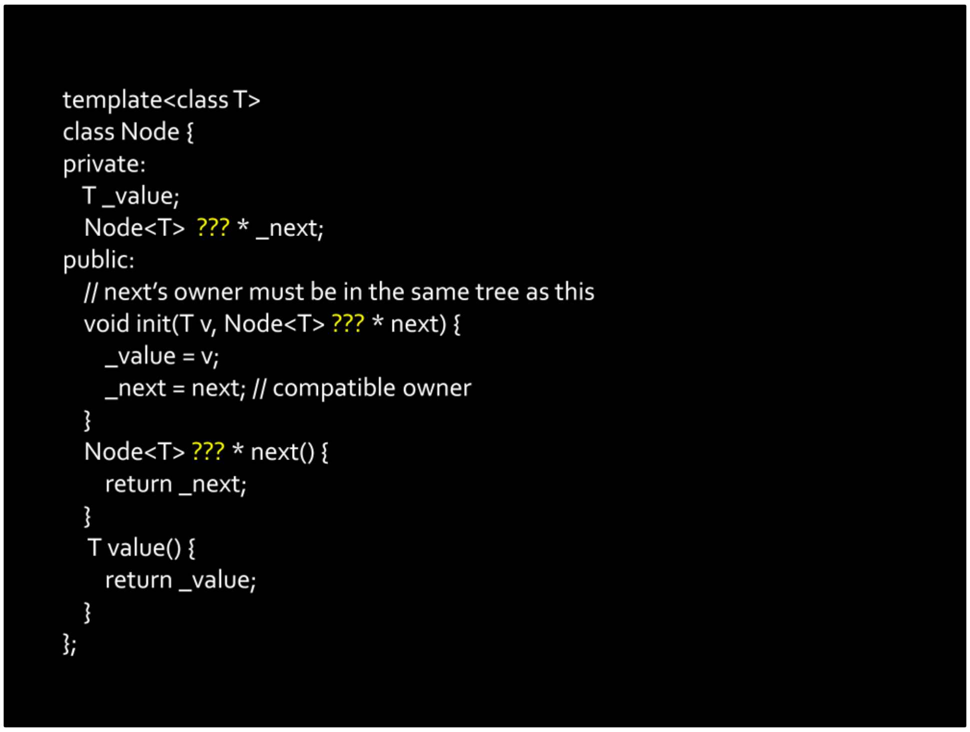
```
template<class T>
class Stack { // Monitor
  mutex _m;
  Node<T> owner(this) * _top;
public:
  void push(T value) {  lock_guard lock(_m);
    Node<T> owner(this) * newNode = new Node<T>;
    newNode->init(value, _top);
    _top = newNode; // same owner
  }
  T pop() {  lock_guard lock(_m);
   if (_top == 0) throw "bad pop";
    Node<T> owner(this) * n = _top->next();
    T value = _top->value();
    _top = n; // same owner!
    delete _top;
    return value;
  }
};

Stack<int> owner(self) * TheStack = new Stack<int>;
```

Stack is a monitor: it contains its own mutex and all methods lock that mutex.
_top is owned by the stack, owner(this). When the class is instantiated, this becomes the instance of the class.
push: new node created with owner(this)
  _top is passed to Node::init, make sure init method of a stack-owned node accepts stack-owned node
   assignment _top = newNode: ownership must match.
pop: assignment _top = _top.next(), make sure _top.next() returns a node whose owner is compatible with the owner of this. Compatible means "from the same ownership tree." In this case _top is owned by stack. next() returns an object owned by _top (see next slide), so they are compatible.

TheStack, being a monitor, is self-owned.



The Stack instance owns the _top node. Each node instance owns the _next node.

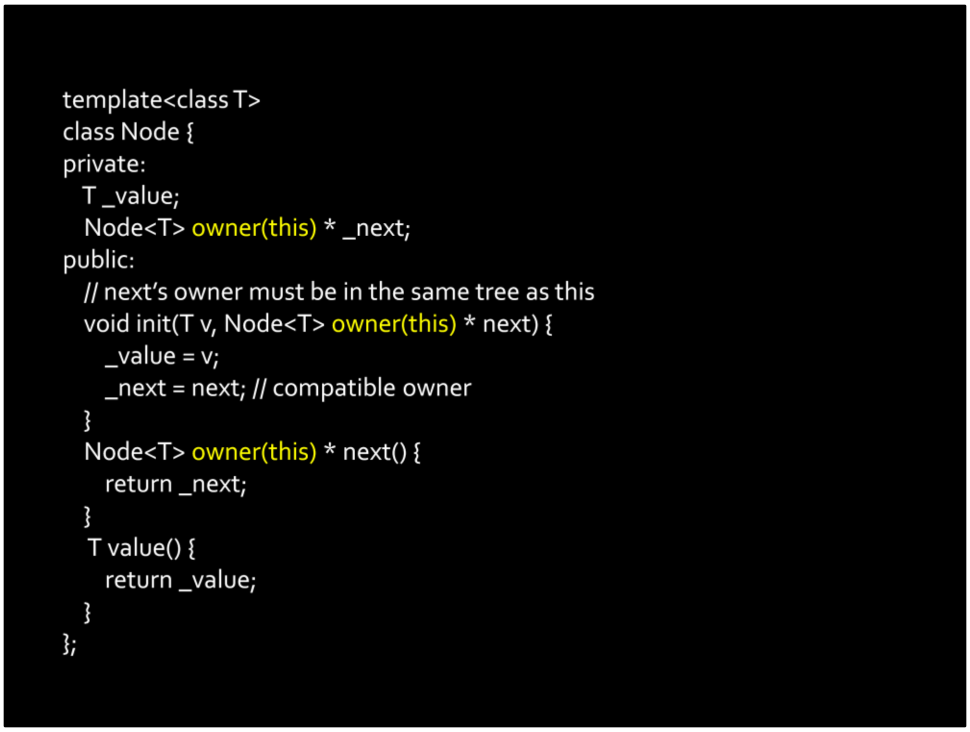


```
template<class T>
class Node {
private:
  T _value;
  Node<T>  ??? * _next;
public:
  // next's owner must be in the same tree as this
  void init(T v, Node<T> ??? * next) {
    _value = v;
    _next = next; // compatible owner
  }
  Node<T> ??? * next() {
    return _next;
  }
  T value() {
    return _value;
  }
};
```

Node owns _next. But we know that the stack owns the top node, so they all belong to the same tree rooted at the stack. They can be cross-linked at will.

init() takes a Node, next, which must be owned by this (the very node). However any node that belongs to the same ownership tree as “this” will work too. Another possible annotation would be owner(of_this).

next() returns a node owned by this. It may be assigned to anything that is a member of the same ownership tree (in our case, it’s the tree rooted at stack).

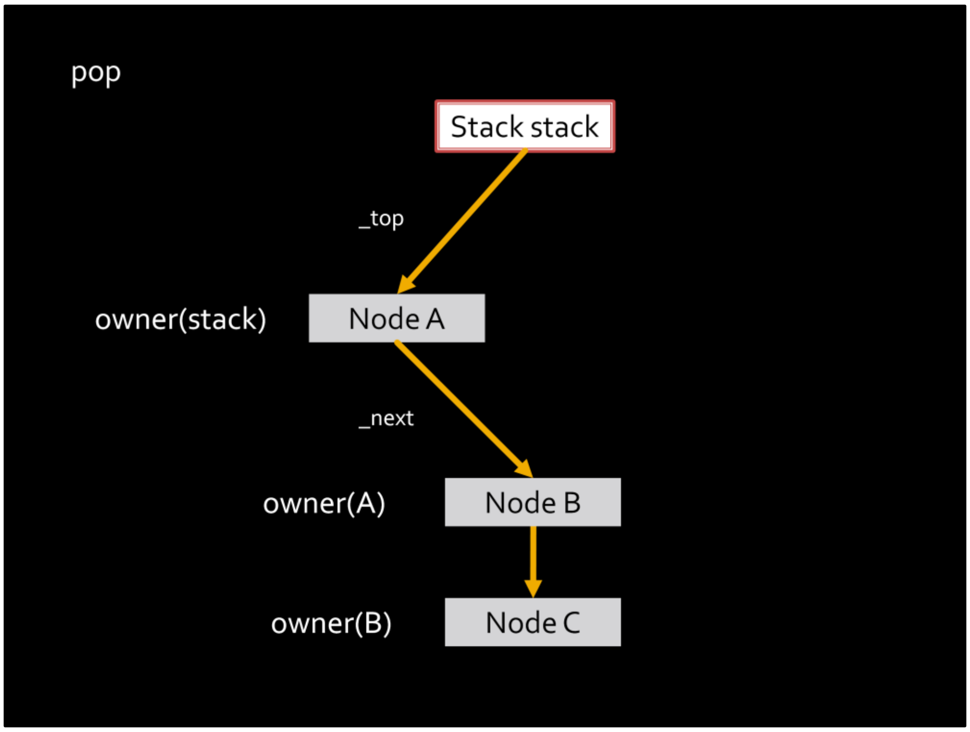


```
template<class T>
class Node {
private:
   T _value;
   Node<T> owner(this) * _next;
public:
   // next's owner must be in the same tree as this
   void init(T v, Node<T> owner(this) * next) {
      _value = v;
      _next = next; // compatible owner
   }
   Node<T> owner(this) * next() {
      return _next;
   }
    T value() {
      return _value;
   }
};
```

Node owns _next. But we know that the stack owns the top node, so they all belong to the same tree rooted at the stack. They can be cross-linked at will.

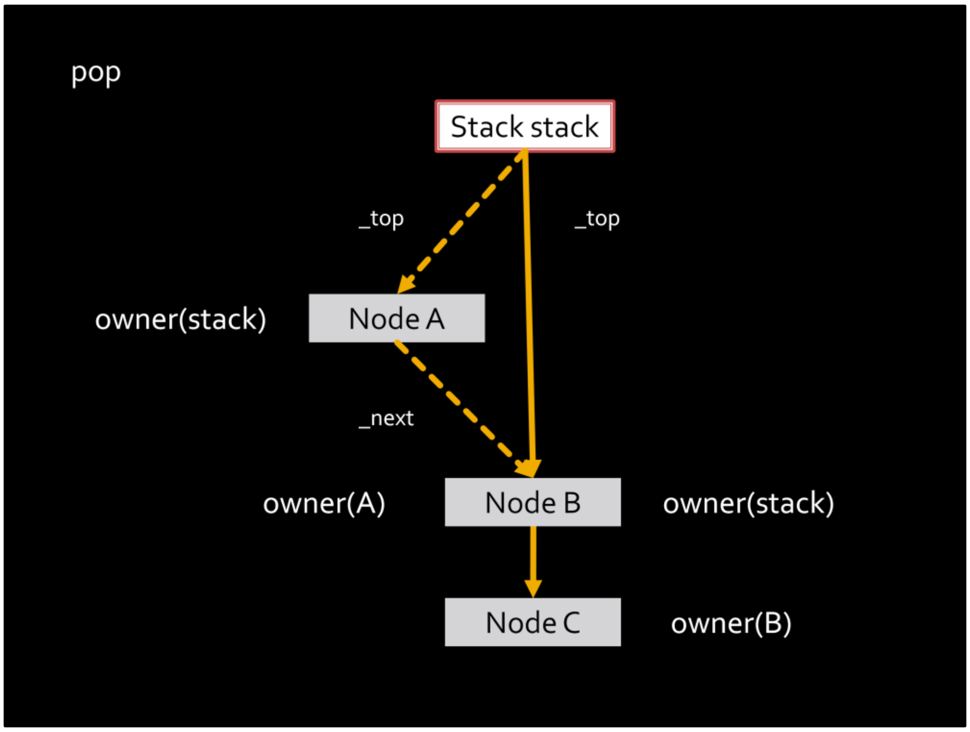
init() takes a Node, next, which must be owned by this (the very node). However any node that belongs to the same ownership tree as “this” will work too. Another possible annotation would be owner(of_this).

next() returns a node owned by this. It may be assigned to anything that is a member of the same ownership tree (in our case, it’s the tree rooted at stack).



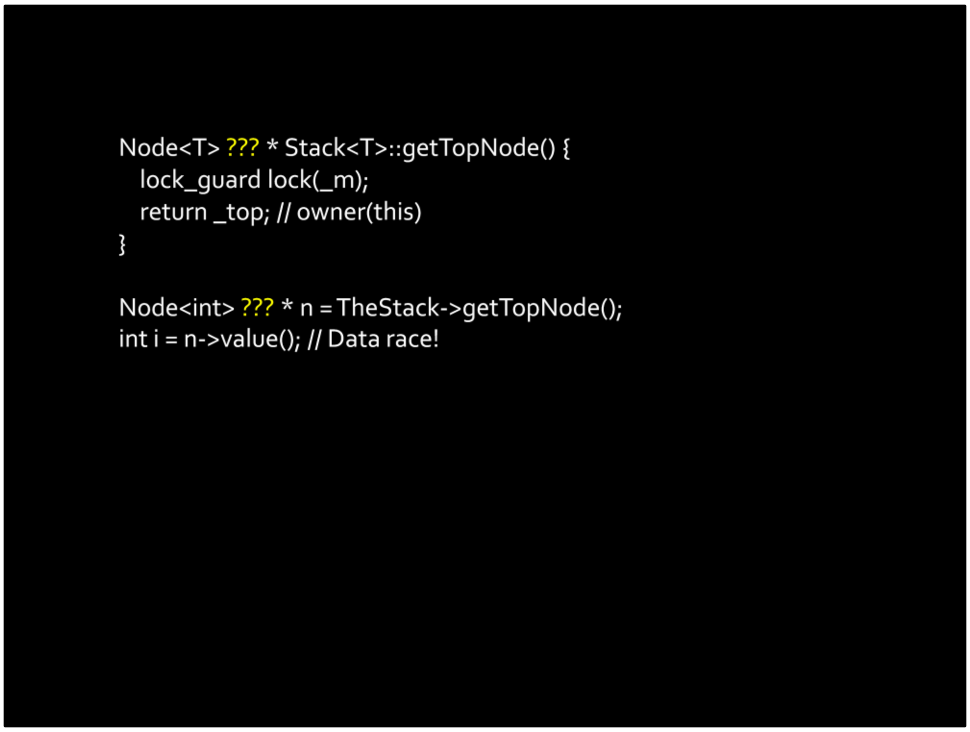
The method pop re-assigns _top, which is a pointer to a node owned by the stack. The new pointer (to Node B) obtained from Node A is owned by Node A. However Node A is owned by the stack, so Node B is in the same tree and can be safely assigned to _top.

I'm stressing these rules of assignment because any breakage will cause the leakage of aliases and to data races.



The method pop re-assigns _top, which is a pointer to a node *owned by* the stack. The new pointer (to Node B) obtained from Node A is *owned by* Node A. However Node A is *owned by* the stack, so Node B is in the same tree and can be safely assigned to _top. As long as the top owner is not changed, such assignment type-checks.

I'm stressing these rules of assignment because any breakage will cause the leakage of aliases and lead to data races.

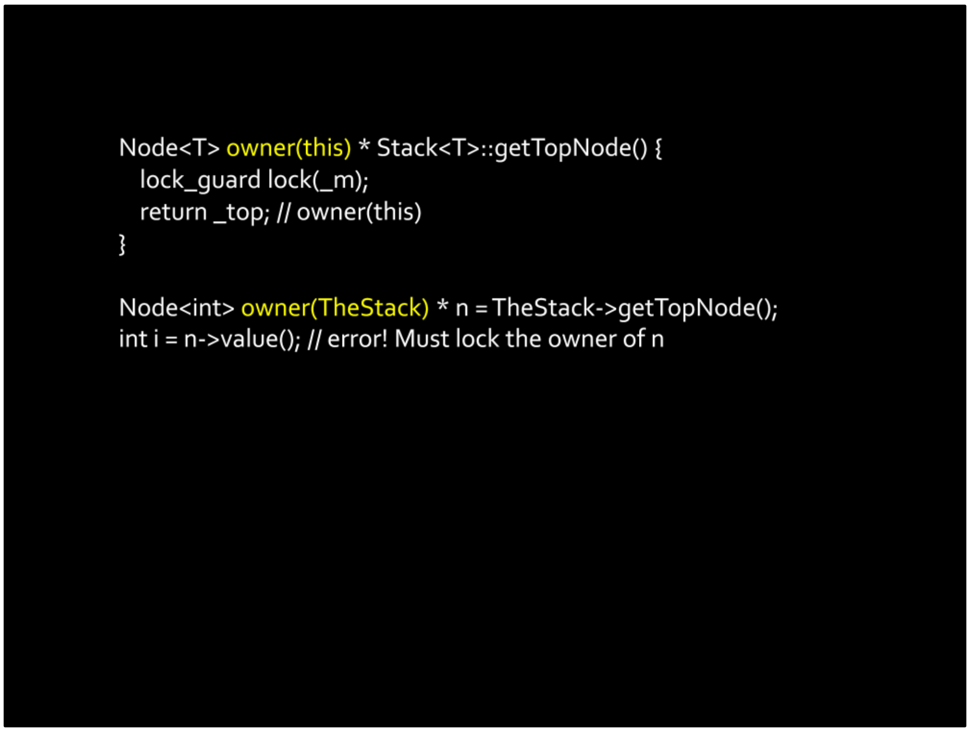


```
Node<T> ??? * Stack<T>::getTopNode() {
   lock_guard lock(_m);
   return _top; // owner(this)
}

Node<int> ??? * n = TheStack->getTopNode();
int i = n->value(); // Data race!
```

Alias control.

Returning an internal alias is dangerous. Correct “typing” helps find the error. Stack::getTop() returns an object owned by the stack.
For the assignment to type-check, the variable that accepts the result of getTop must be owner(stack). This variable cannot be accessed without locking the stack (remember: the top of the ownership tree must be locked).

In general, avoid returning aliases to owned objects. Feel free to return aliases to self-owned objects though.

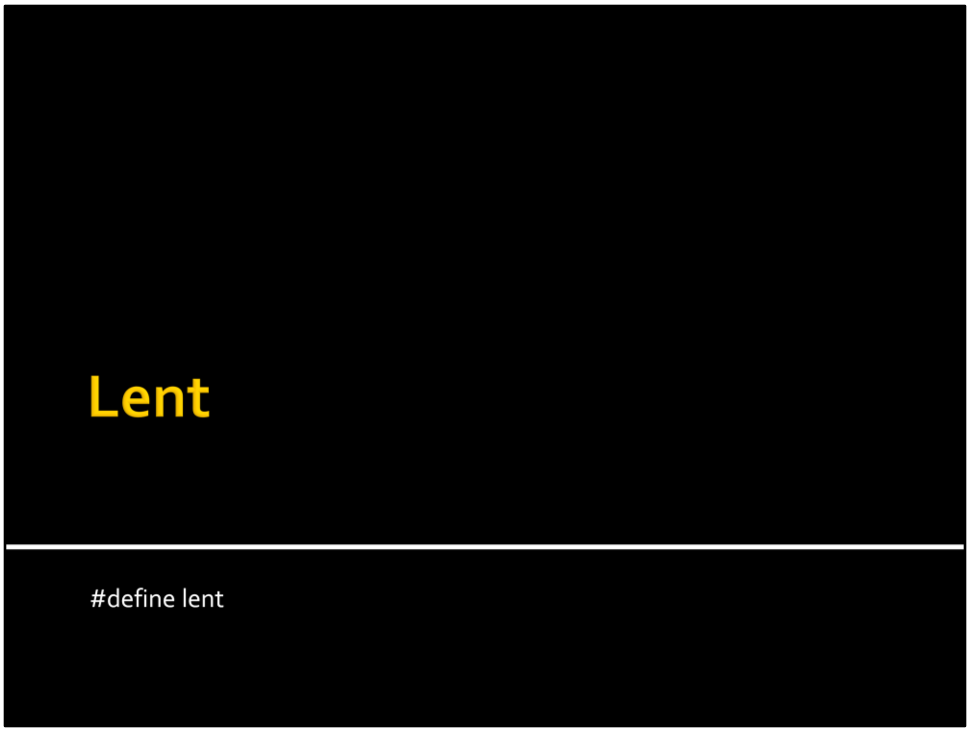


```
Node<T> owner(this) * Stack<T>::getTopNode() {
  lock_guard lock(_m);
  return _top; // owner(this)
}

Node<int> owner(TheStack) * n = TheStack->getTopNode();
int i = n->value(); // error! Must lock the owner of n
```

Alias control.

Returning an internal alias is dangerous. Correct “typing” helps find the error.
Stack::getTop() returns an object owned by the stack.
For the assignment to type-check, the variable that accepts the result of getTop must be owner(stack). This variable cannot be accessed without locking the stack (remember: the top of the ownership tree must be locked).

In general, avoid returning aliases to owned objects. Feel free to return aliases to self-owned objects though.



Lent

#define lent

We worry about passing owned objects to functions or methods of other objects because of the danger that they will squirrel the alias and then access it without locking the owner.

Take for instance

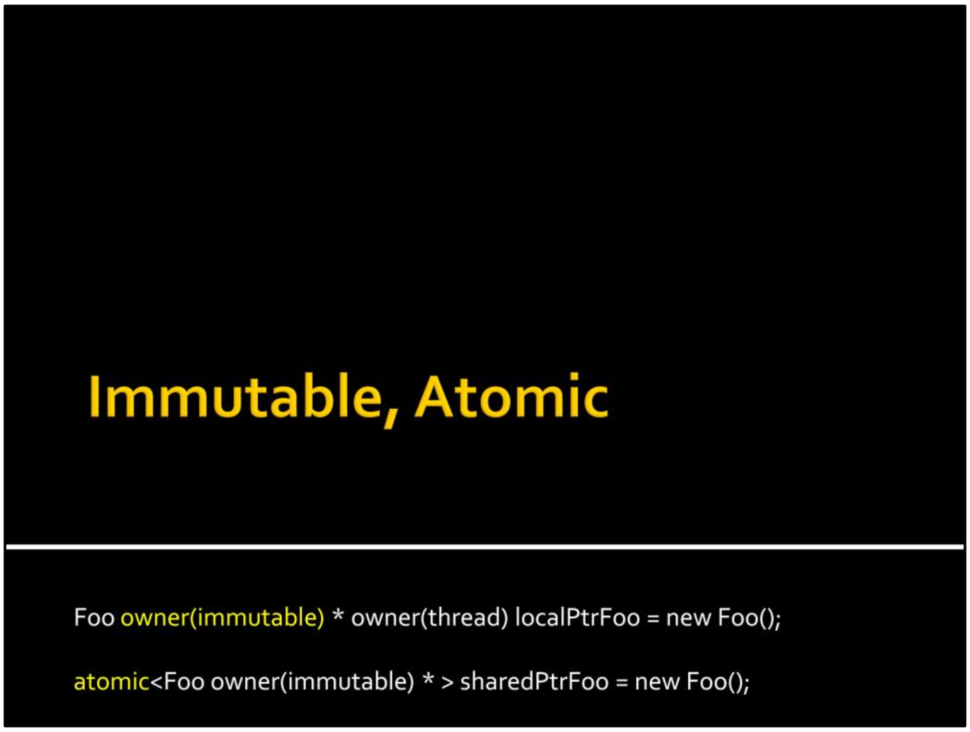
strlen(char const * str);

We'd like to be able to call it with any string owned by a monitor. We know that strlen doesn't squirrel away the string we pass to it. We'd like to be able to express it in the system. Hence the new annotation "lent," which means "I promise not to store this reference." So the correct signature should be:

strlen (char const lent * str);

In general, when an object is lent, no aliases to its minions may be stored, and no external aliases may be stored inside a lent object.

In practice, we don't annotate Standard Library functions, but we do annotate user-defined functions/methods.



Immutable, Atomic

```
Foo owner(immutable) * owner(thread) localPtrFoo = new Foo();

atomic<Foo owner(immutable) * > sharedPtrFoo = new Foo();
```

The sharing of immutable data is safe. Owner(immutable) starts a new tree. All its branches must be immutable too, except when they are self-owned.

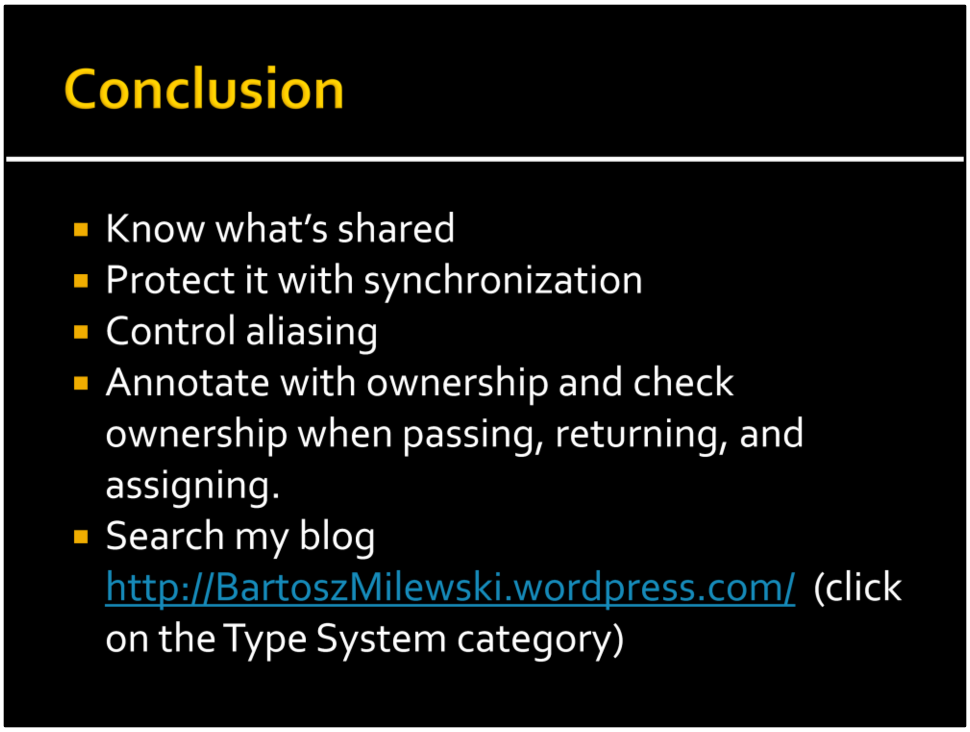
Publication safety: After immutable data is constructed, it is published by assigning it to a pointer. If this pointer is shared and not owned by a monitor, it must be *atomic*. Atomic is shallow and assumes sharing. An atomic pointer must point to a shareable object. Here the object is shareable by way of immutability. It may also be a monitor.

Unique

Unique (a.k.a., linear) types disallow aliases. In C++ they are modeled as unique_ptr.

Unique objects may be safely passed between threads, because they are never shared.

Programmer must make sure that when unique_ptr is created, no aliases to the object or its minions are squirreled away. Any aliases created through unique_ptr::get() must be *lent* only.



Conclusion

- Know what's shared
- Protect it with synchronization
- Control aliasing
- Annotate with ownership and check ownership when passing, returning, and assigning.
- Search my blog http://BartoszMilewski.wordpress.com/ (click on the Type System category)

There is a series of blogs describing in some detail the race-free type systems. You'll also find there links to literature.

http://bartoszmilewski.wordpress.com/category/type-system/