

# CS 541 6 Recitation 5

## multi-threading II

---

09/26/2025

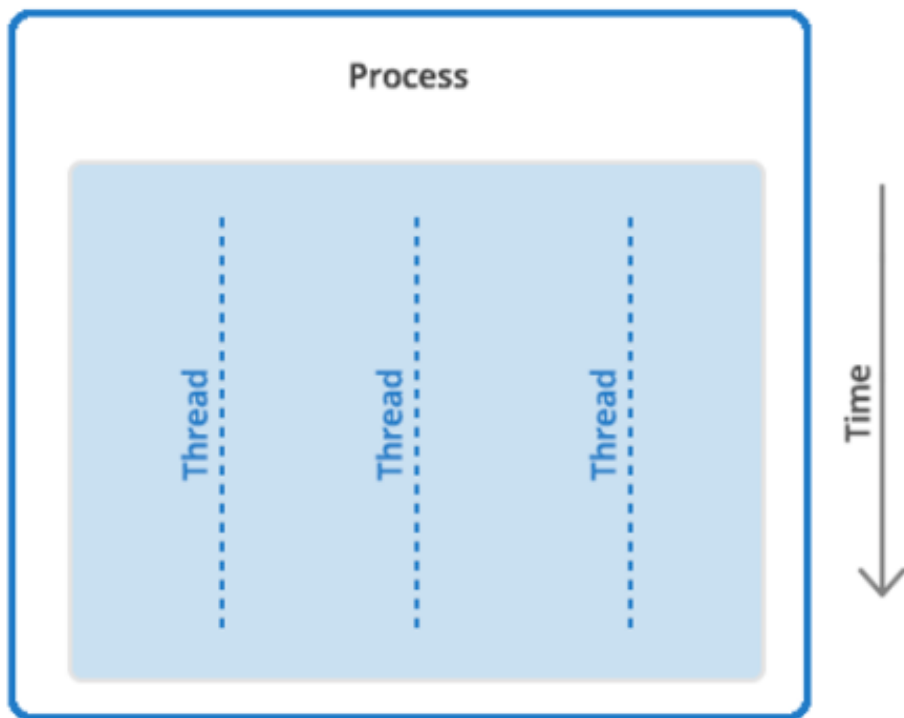
Jamal Hashim

# Recap

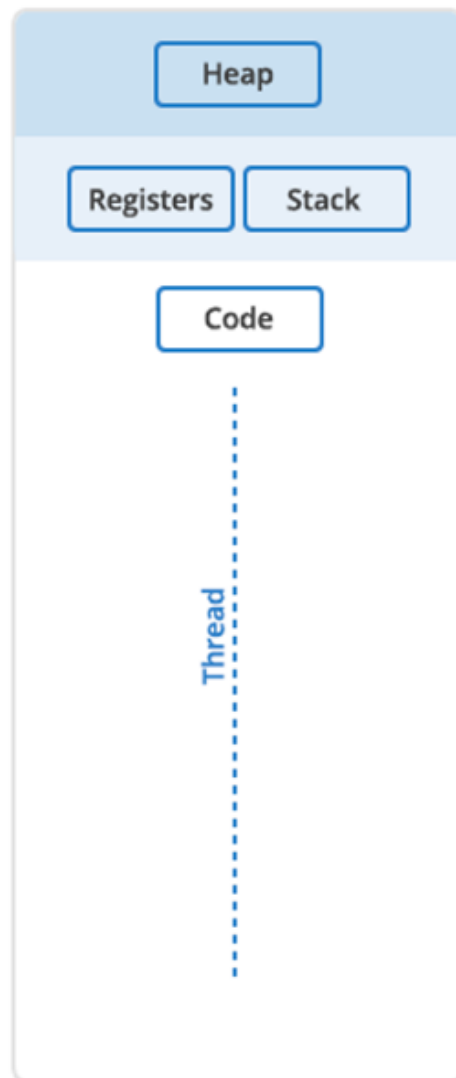
---

- Multithreading
- Race condition

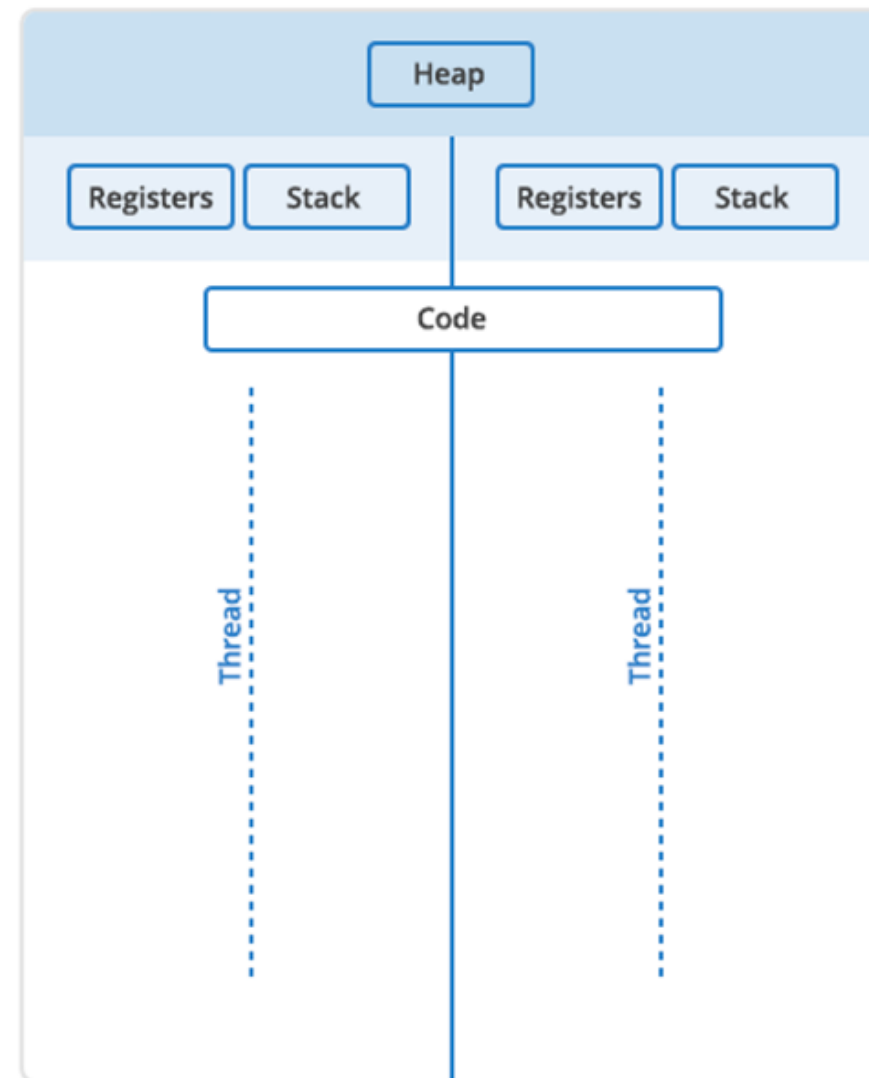
# Concurrency



Single Thread



Multi Threaded



# Sharing data among threads

---race condition

- Race condition:
  - The situation 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.

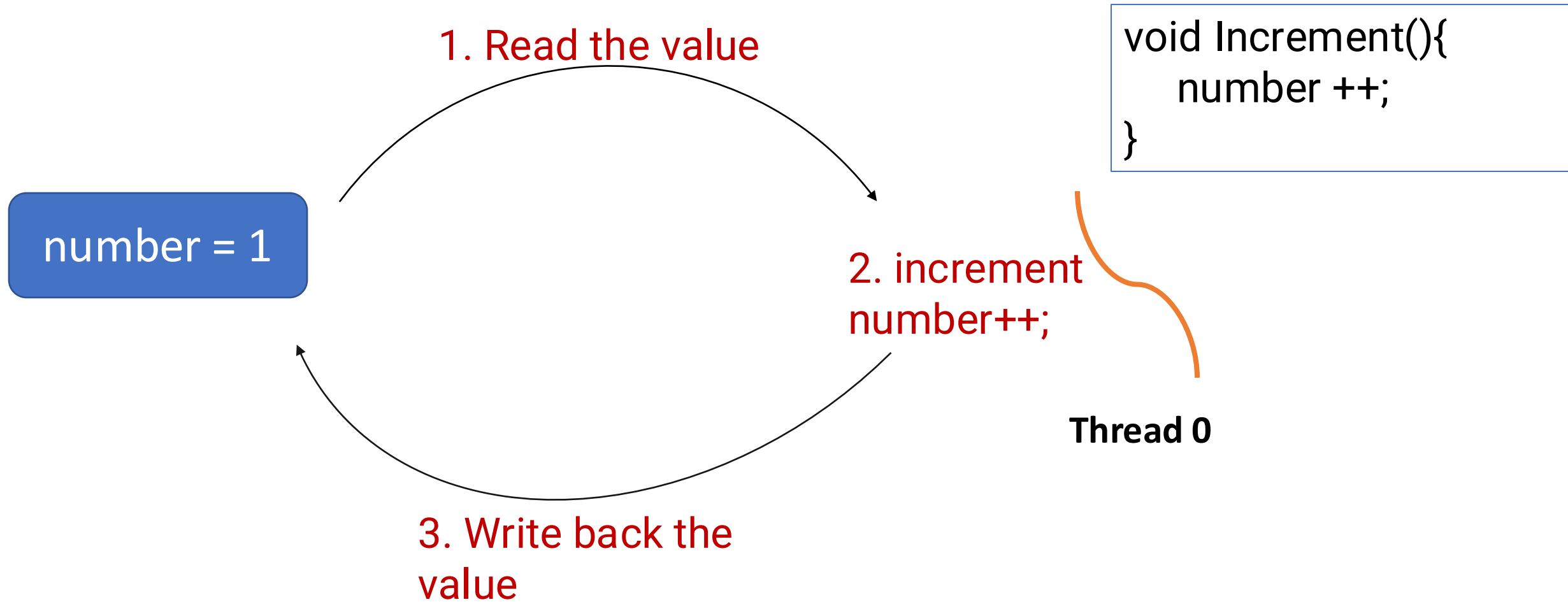
# Sharing data among threads

---race condition

- Example: Concurrent increments of a shared integer variable.
  - Each thread shares an integer called count initialized to 0, increments it 1 million times concurrently without any synchronization

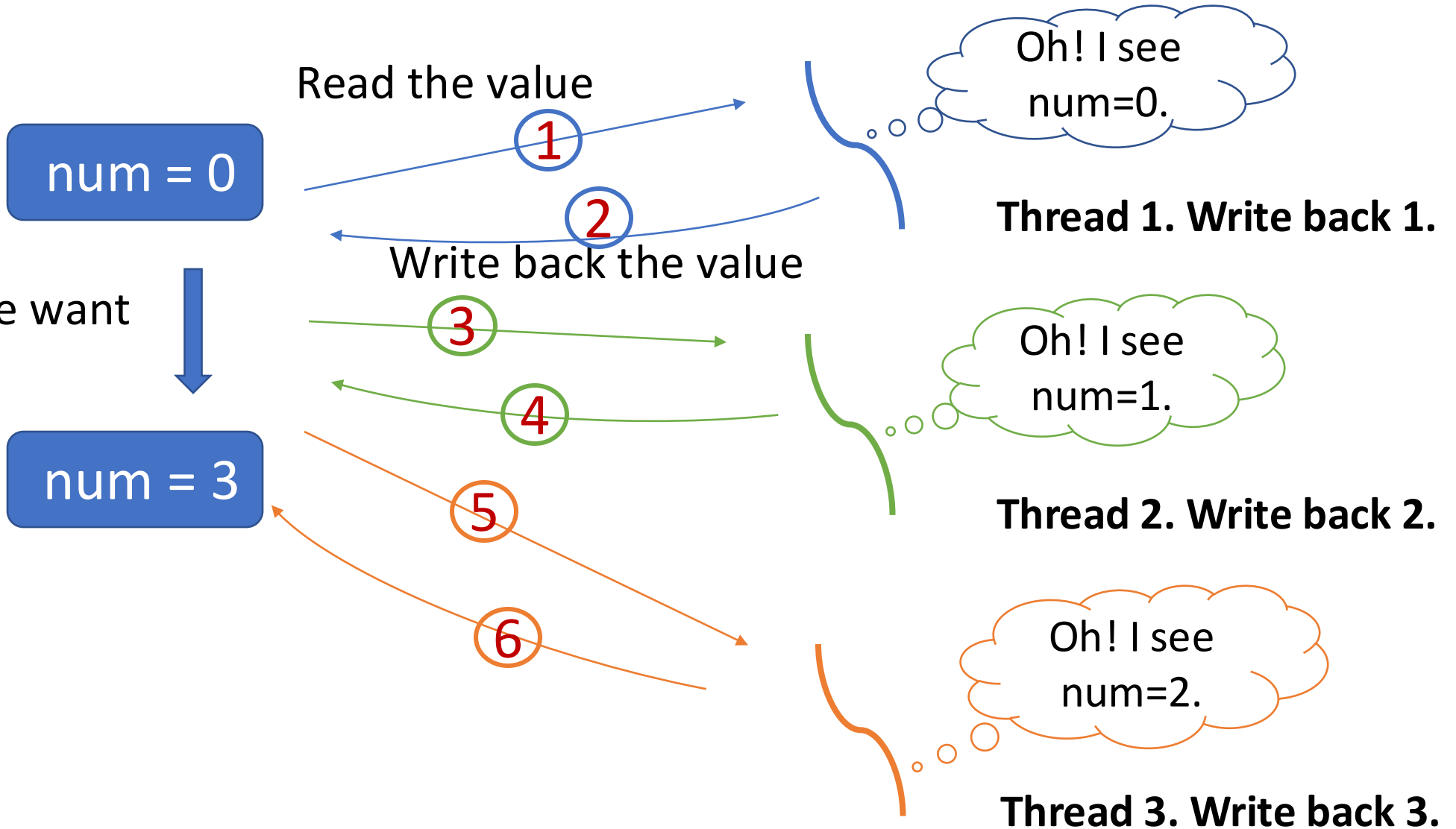
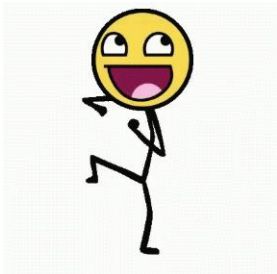
Number of threads	Final value
1	1000000
2	1059696
3	1155035
4	1369165

## Example: Concurrent increments of a shared integer variable

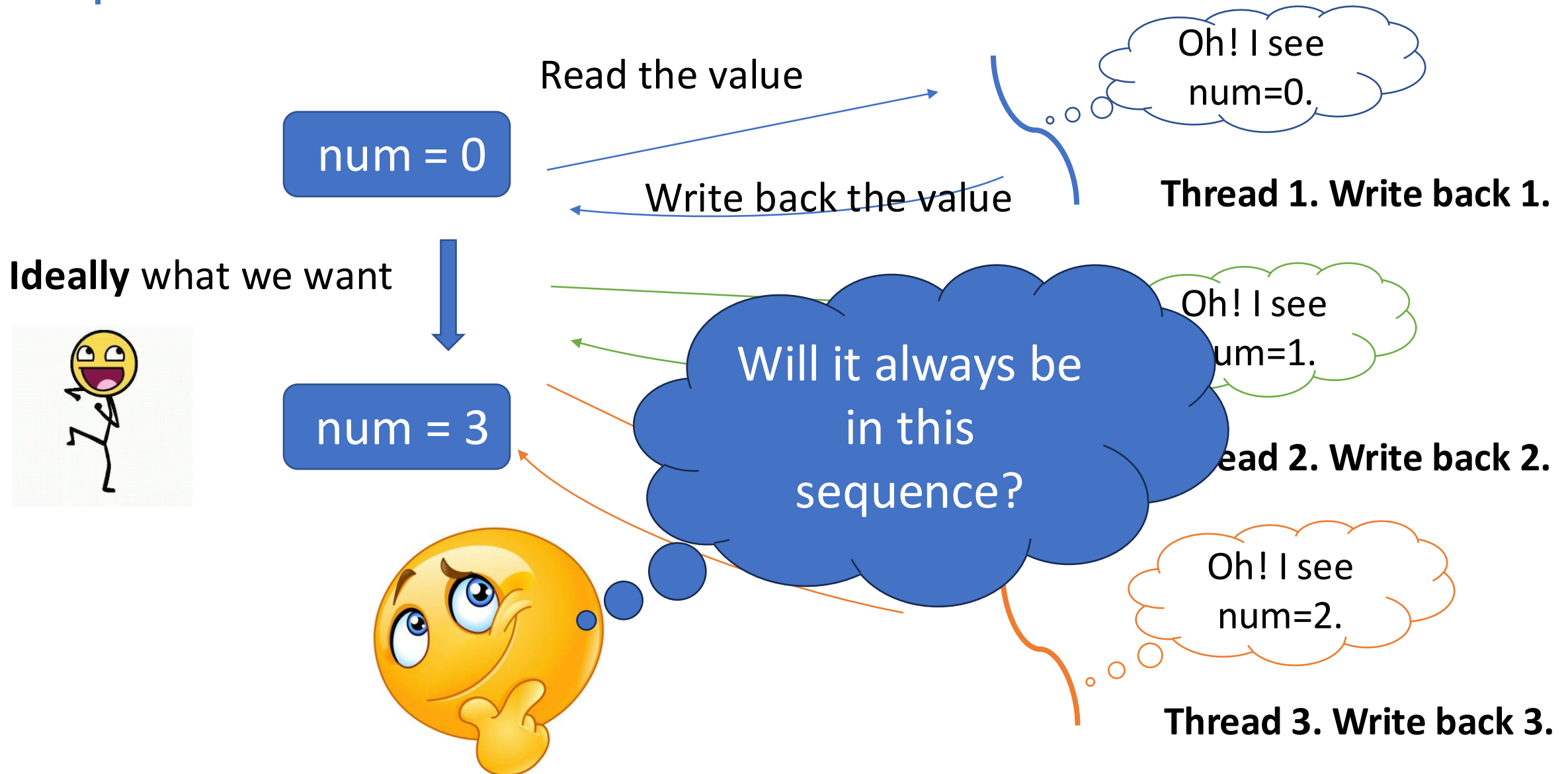


# Example: Concurrent increments of a shared integer variable

Ideally what we want

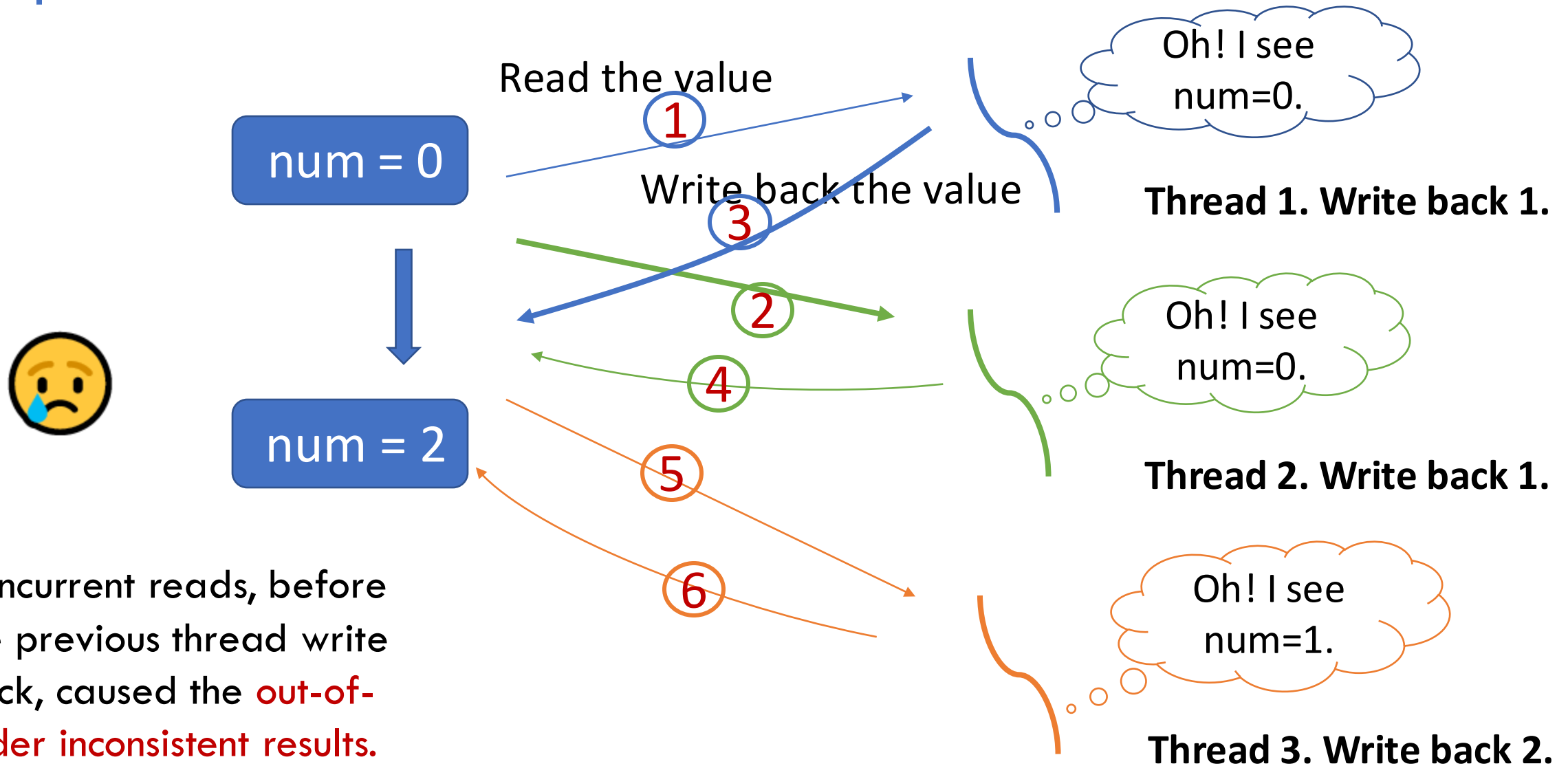


# Example: Concurrent increments of a shared integer variable





# Example: Concurrent increments of a shared integer variable



# Thread Safety

- A function, a piece of code, or an object is **thread-safe** when it can be **invoked** or **accessed** **concurrently** by **multiple threads** **without** causing unexpected behavior, race conditions, or data corruption.

# Thread safe

---

- Entities in C++ standard library and their thread-safety guarantees

# Thread safe?

- Is integer type inherently thread-safe?
  - No, as we showed just now



How to make it  
thread-safe?

# std::atomic

- A template that defines an **atomic** type.



```
template< class T >  
struct atomic;
```

(1)

(since C++11)

```
template< class U >  
struct atomic<U*>;
```

(2)

(since C++11)

```
template< class U >  
struct atomic<std::shared_ptr<U>>;
```

(3)

(since C++20)

```
template< class U >  
struct atomic<std::weak_ptr<U>>;
```

(4)

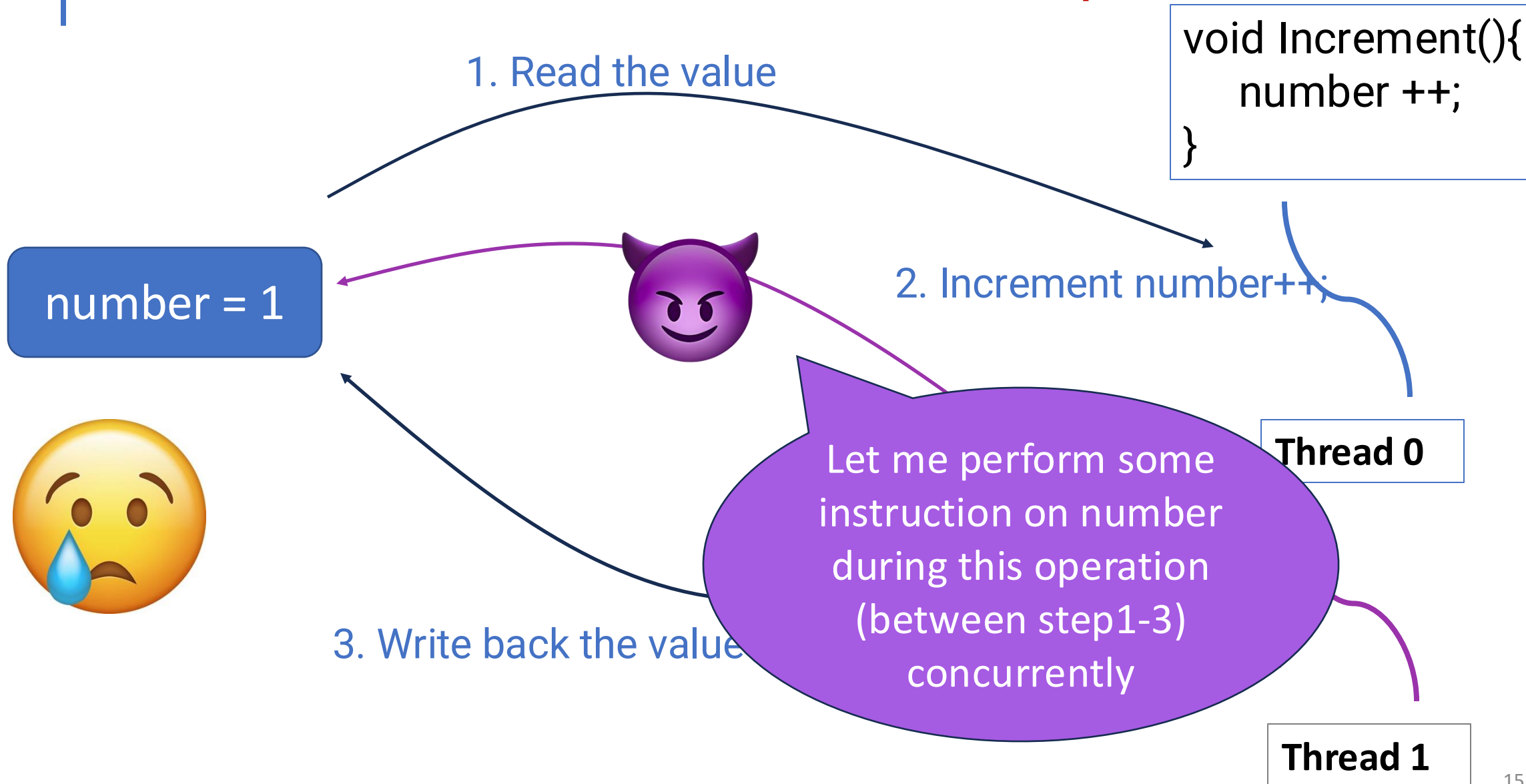
(since C++20)

\*  
(more at  
the end of  
recitation if  
have time)

# Atomic

- An atomic operation is an **indivisible** operation.
- The operation is **either done or not done**. Such an operation would **never be half-done** from any thread in the system.

# Data race condition: non-atomic access pattern



# Data race condition: non-atomic access pattern



Add... More

Support diversity in C++ with `#include <C++>`

intel PC-lint Solid Sands

Share

Policies

Other

C++ source #1

x86-64 gcc 11.2 (C++, Editor #1, Compiler #1)

x86-64 gcc 11.2

Compiler options...

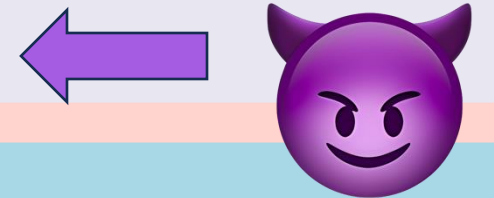
(3, 14)

C++

Output... Filter... Libraries Add new... Add tool...

```
1 int main()
2 {
3     volatile int val = 0;
4     val ++;
5     return val;
6 }
```

```
1 main:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], 0
5     mov     eax, DWORD PTR [rbp-4]
6     add     eax, 1
7     mov     DWORD PTR [rbp-4], eax
8     mov     eax, DWORD PTR [rbp-4]
9     pop     rbp
10    ret
```

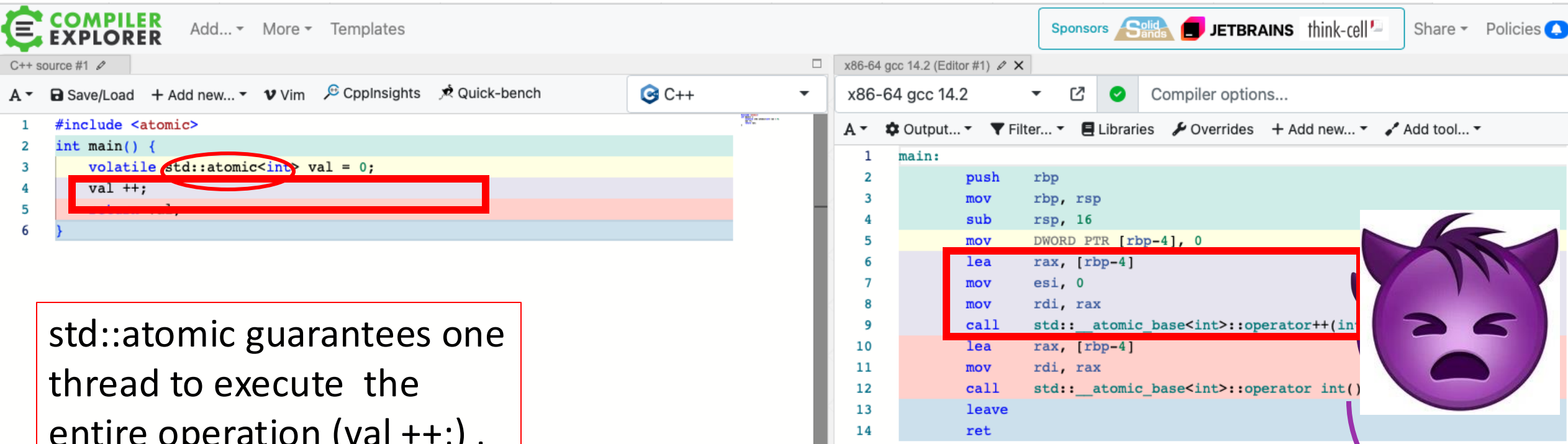


```
std::thread t1([&val]() {
    val++;
});
```

Another concurrent thread t1



# Atomic access



The screenshot displays the Compiler Explorer interface. On the left, the C++ source code is shown with the following content:

```
1 #include <atomic>
2 int main() {
3     volatile std::atomic<int> val = 0;
4     val++;
5 }
6
```

The `std::atomic<int>` type and the `val++` operation are highlighted with a red box. On the right, the assembly output for x86-64 gcc 14.2 is shown:

```
1 main:
2     push    rbp
3     mov     rbp, rsp
4     sub     rsp, 16
5     mov     DWORD PTR [rbp-4], 0
6     lea     rax, [rbp-4]
7     mov     esi, 0
8     mov     rdi, rax
9     call    std::__atomic_base<int>::operator++(int*)
10    lea     rax, [rbp-4]
11    mov     rdi, rax
12    call    std::__atomic_base<int>::operator int()
13    leave
14    ret
```

The assembly instructions corresponding to the `val++` operation (lines 6-9) are highlighted with a red box. A purple cat face emoji is placed next to the assembly output.

`std::atomic` guarantees one thread to execute the entire operation (`val++`), during which no other thread interfering or interrupting

```
std::thread t1([&val]() {
    val++;
});
```

Another concurrent thread t1

# Atomic

- An atomic operation is an **indivisible operation**.
- `std::atomic` are **implemented** using hardware supports provided by modern CPU:
  - Examples of **atomic instructions**:
    - Compare-and-Swap (CAS)
    - Load-Linked/Store-conditional (LL/SC)
    - `fetch_and_add` (FAA)
  - **Different CPUs** provide **different sets of atomic instructions**. The implementation of `std::atomic` varies from architecture to architecture

# Atomic member functions

- Atomic type: `std::atomic<type>`
- Constructor `std::atomic<bool> x(true);` `std::atomic<uint32_t> y(0);`
- `store()` `x.store(false);` `y.store(1, std::memory_order_relaxed);`

## More atomic member functions

- `load()`
- `exchange()`
- `operator=`
- `operator+=`, `operator -=`
- `operator++`, `operator--`

```
bool z = x.load();
```

```
uint32_t m = y.exchange(100);
```

```
y = 2;
```

```
y += 1;    y.fetch_add(1);
```

(since C++20)

```
y ++;
```

What about `y = y + 1`?

# More atomic member functions

- `load()`  
`bool z = x.load();`
- `exchange()`  
`uint32_t m = y.exchange(100);`
- `operator=`  
`y = 2;`
- `operator+=`, `operator -=`  
`y += 1;    y.fetch_add(1);`
- `operator++`, `operator--`  
`y ++;`

What about `y = y + 1`?

When multithreading, leads to **race condition**, because it involves multiple operations (read x, +1 and then assignment operation)

# Thread safe

---

- `std::atomic`
- `std::shared_ptr`

## `std::vector`

- Does `std::vector` guarantee thread-safety?

# Multithreads' data sharing with `std::vector`

- When is `std::vector` thread-safe?
  - Each thread has its own instance of `std::vector` (no concurrency)
  - Read-only access
- When is `std::vector` not thread-safe?
  - Simultaneous Read and Write
  - Concurrent modification
  - Reallocation access on reallocation or modification



# Read-only-access of std::vector



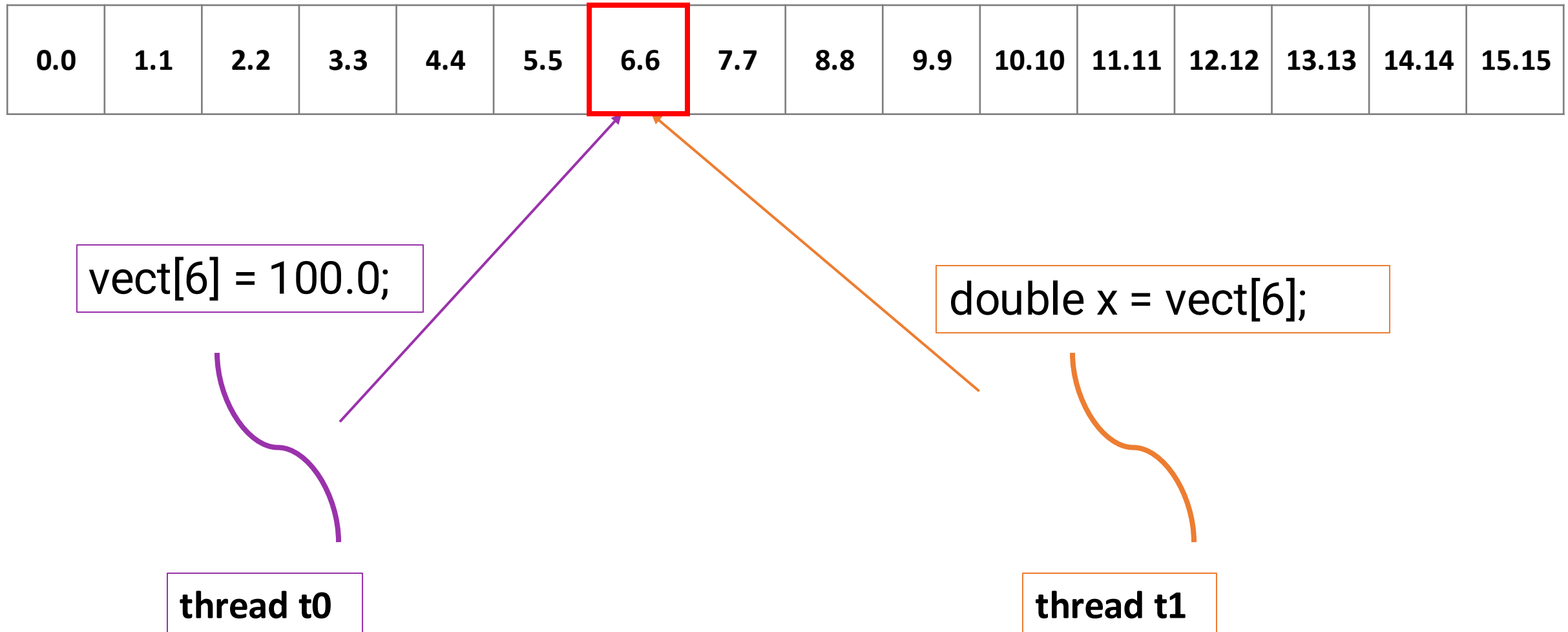
```
void read_vector(const std::vector<double>& vec, int thread_id, double& sum) {  
    for (const auto& value : vec) {  
        sum += value;  
    }  
}
```

// Each thread reads the vector and accumulates the sum

Thread safe, because only  
concurrent reads

```
int main() {  
    std::vector<double> vec(100, 1.00);  
    double t1_sum;  
    double t2_sum;  
    std::thread t1(read_vector, std::ref(vec), 1, std::ref(t1_sum));  
    std::thread t2(read_vector, std::ref(vec), 2, std::ref(t2_sum));  
    t1.join();  
    t2.join();  
    std::cout << "t1_sum=" << t1_sum << ", t2_sum=" << t2_sum;  
    ...}  
}
```

# Simultaneous read and write



# Simultaneous read and write

Concurrent Read+write  
to the **SAME** element  
is **NOT** thread-safe

0.0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.10	11.11	12.12	13.13	14.14	15.15
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-------	-------	-------	-------	-------	-------

`vect[6] = 100.0;`

thread t0

`double x = vect[6];`

thread t1

x could be  
6.6 or 100.0  
after this.

# Concurrent access with reallocation



Is this code  
thread-safe?

```
void add_elements(std::vector<int>& vec, int thread_id) {  
    for (int i = 0; i < 10; ++i) {  
        vec.push_back(i);  
    }  
}
```

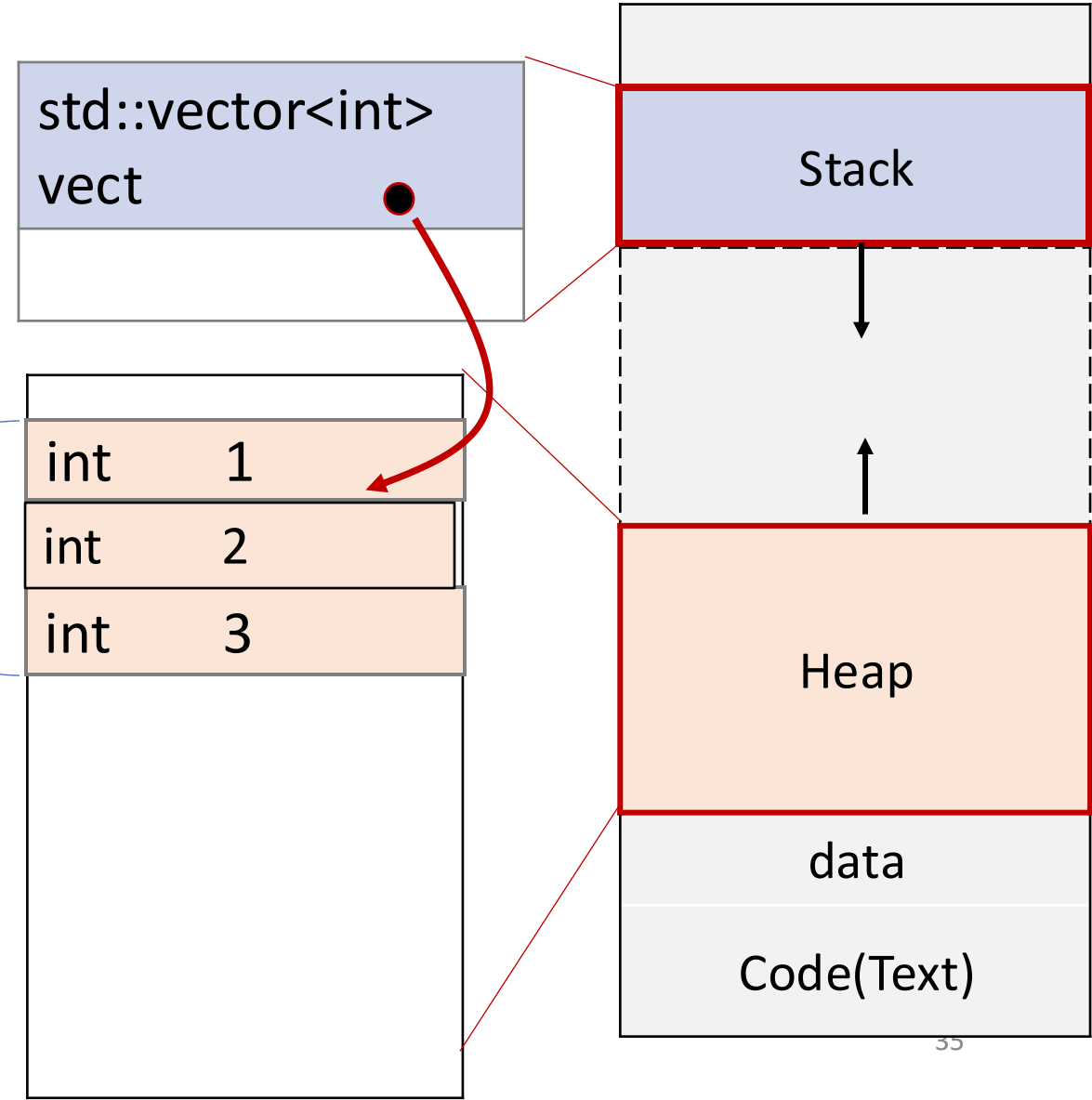
```
int main() {  
    std::vector<int> vec = {1, 2, 3};  
    std::thread t1(writer, std::ref(vec));  
    std::thread t2([&vec]() {  
        std::cout << "value: " << vec.back(); << std::endl;});  
  
    t1.join();  
    t2.join();  
    ...}
```

# How is `std::vector` allocated in memory

```
int main(){  
    std::vector<int> vect= {1,2,3};  
  
}
```

`main()`

Suppose  
`size() = capacity() = 3`

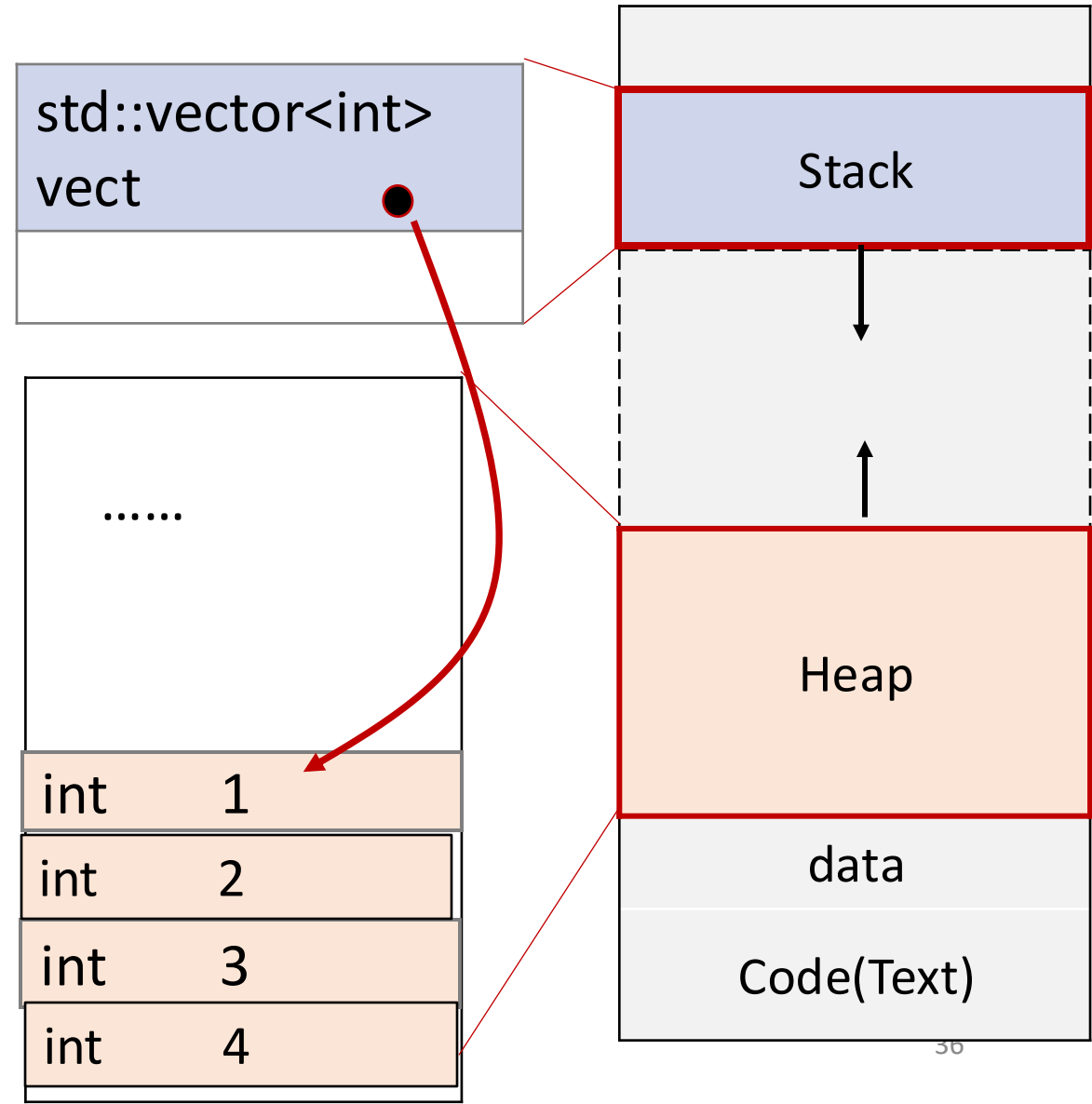


# How is `std::vector` allocated in memory

```
int main(){  
    std::vector<int> vect= {1,2,3};  
    vect.push_back(1);  
}
```

- after the operation the new `size()` is greater than old `capacity()`,
- a **reallocation** takes place
  - all iterators and all references to the elements are invalidated.

`main()`



# Concurrent access with reallocation

```
void add_elements(std::vector<int>& vec, int thread_id) {  
    for (int i = 0; i < 10; ++i) {  
        vec.push_back(i);  
    }  
}
```

Not thread safe:  
one thread is modifying the  
std::vector (push\_back), while  
another thread reads from it  
(back()), there's a risk of data  
races.

```
int main() {  
    std::vector<int> vec = {1, 2, 3};  
    std::thread t1(writer, std::ref(vec));  
    std::thread t2([&vec]() {  
        std::cout << "value: " << vec.back(); << std::endl;});  
  
    t1.join();  
    t2.join();  
    ...}
```

# Locking

---



# Locking

---protecting data with mutex



- How does mutex work?
  - Before accessing a shared data structure, you **lock the mutex** associated with that data
  - When finished accessing the data structure, you **unlock the mutex**.



# std::mutex



exclusive, non-recursive ownership

- A thread owns the **mutex** from the time when it call **lock()** until it calls **unlock()**
- 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.

# Locking

---std::mutex::lock(), unlock()

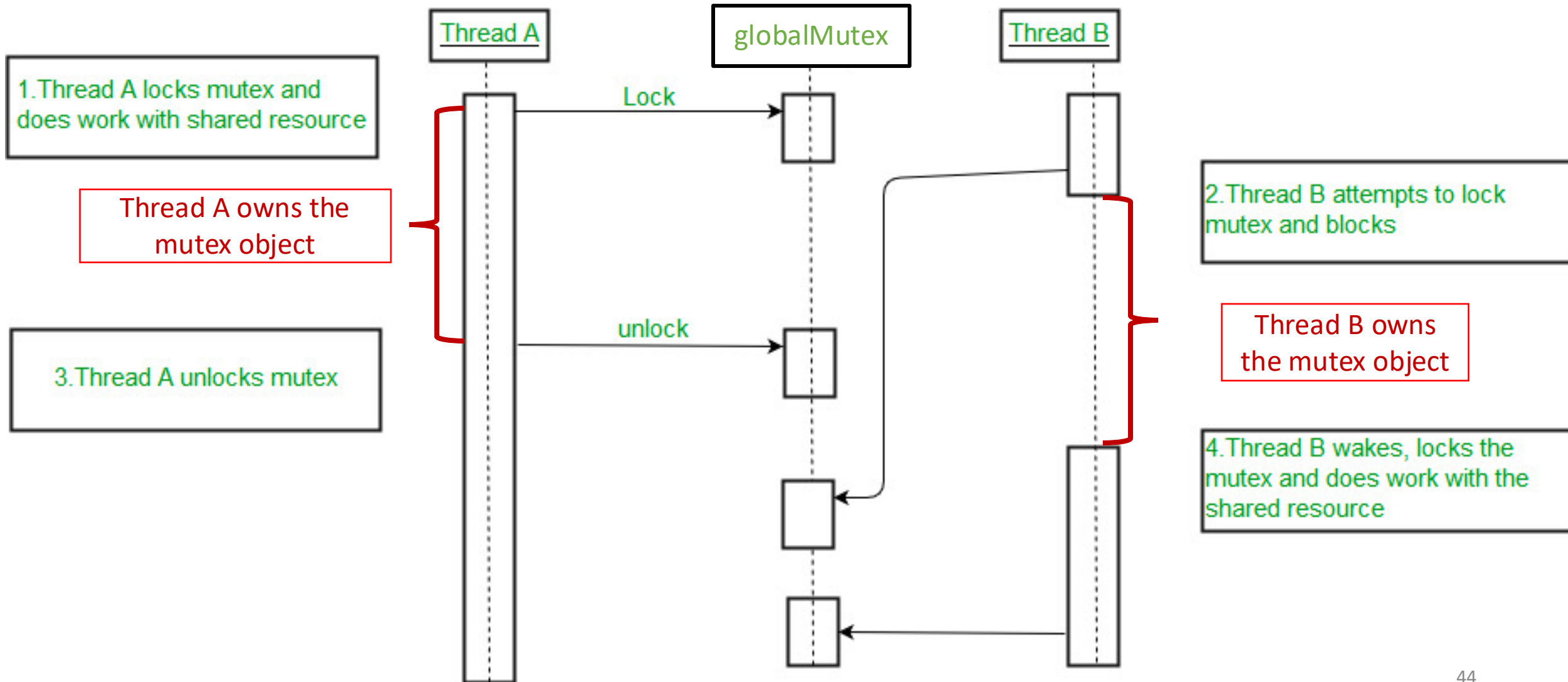
```
1  int    global_num = 0;
2  std::mutex    globalMutex;

3  void incre(int num){
4      globalMutex.lock();
5      global_num = global_num + 1;
6      globalMutex.unlock();
7  }

8  int main(){
9      std::thread threadA(incre, 10);
10     std::thread threadB(incre, 10);
11     threadA.join();
12     threadB.join();
...}
```

Only one  
thread could  
enter line 5 at  
a time

# Mutex and Lock in C++



# Locking

---std::mutex::lock(), unlock()

```
int    global_num = 0;
std::mutex    globalMutex;

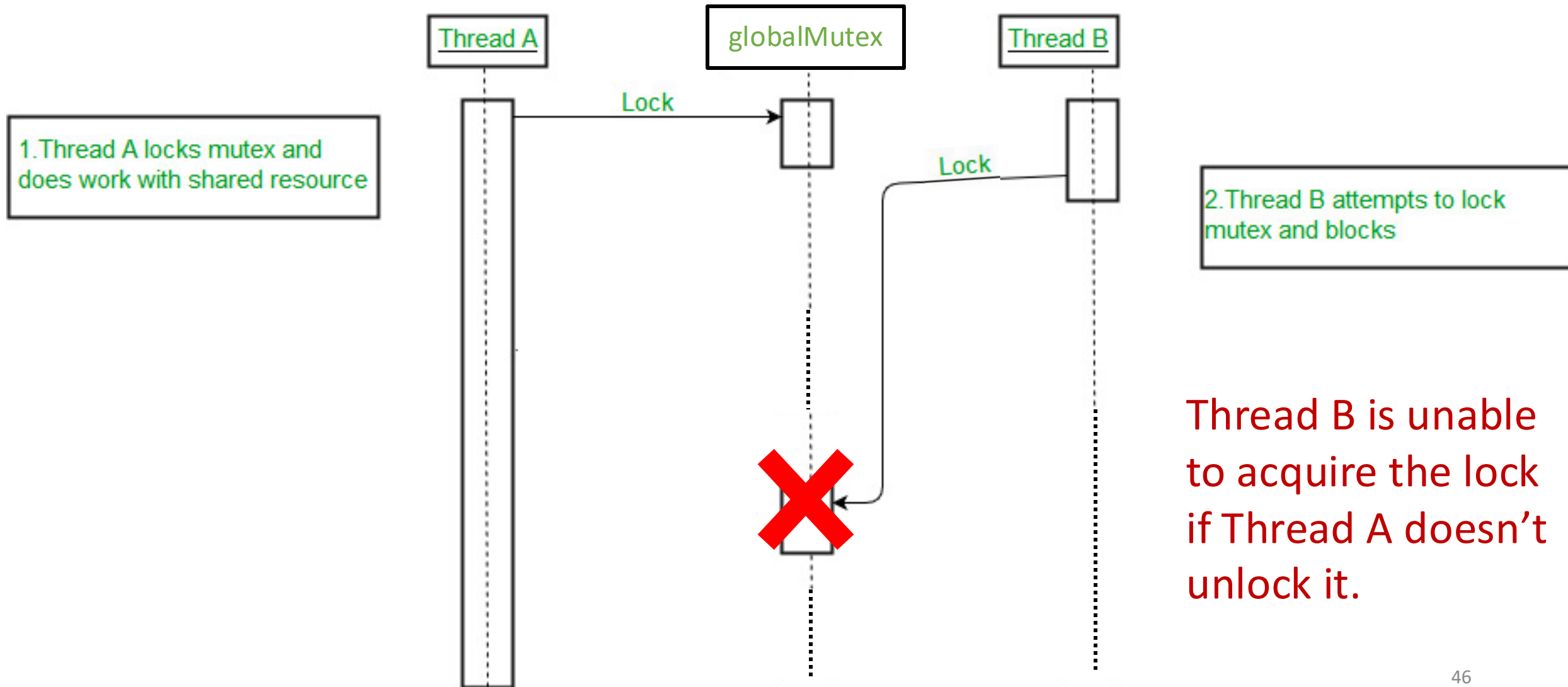
void incre(int num){
    globalMutex.lock();
    global_num = global_num + 1;
    globalMutex.unlock();
}

int main(){
    std::thread threadA(incre, 10);
    std::thread threadB(incre, 10);
    threadA.join();
    threadB.join();
}
```

Now, what will happen, if I forget to call mutex.unlock()?



# Mutex and Lock in C++



# Mutex and Lock in C++



- A Mutex is a **lock** that we set **before** using **a shared resource** and **release after using it**.
- When the lock is set by one thread, then **no other thread** can access the locked region of code.
- Mutex lock could **only be released** by the **thread who locked it**.

# Locking

---std::mutex::lock(), unlock()

- std::mutex::lock(), unlock()
  - It is **not recommended** practice to call lock(), unlock() directly, because this means that you have to remember to call **unlock()** on **every code path out of a function that called lock()**, including those due to exceptions.



# RAII (Resource Acquisition is initialization) re-visit

- Resource acquisition must succeed for initialization to succeed:
  - In RAII, holding a resource is a class invariant is tied to object lifetime: resource allocation is done during object creation, by the constructor; while resource deallocation is done during object destruction, by the destructor.
- If there are no object leaks, there are no resource leaks.
  - The resource is guaranteed to be held between when initialization finishes and finalization starts, and to be held only when the object is alive.

# RAII (Resource Acquisition is initialization)

*// problem #1*

```
{  
    int *arr = new int[10];  
}  
// arr goes out of scope but we didn't delete it, we now have a memory leak 😞
```

*// problem #2*

```
{  
    std::thread t1( [] () {  
        // do some operations  
    });  
}  
// thread t1 is created but not joined, if it goes out of scope, std::terminate is  
called, this implementation doesn't properly handle the thread's life cycle 😞
```

*// problem #3*

```
Std::mutex globalMutex;  
Void func() {  
    globalMutex.lock();  
}  
// if we never unlocked the mutex(or exception occurred before unlock),  
it will cause a deadlock when other thread tries to acquire this lock 😞
```

# RAII (Resource Acquisition is initialization)

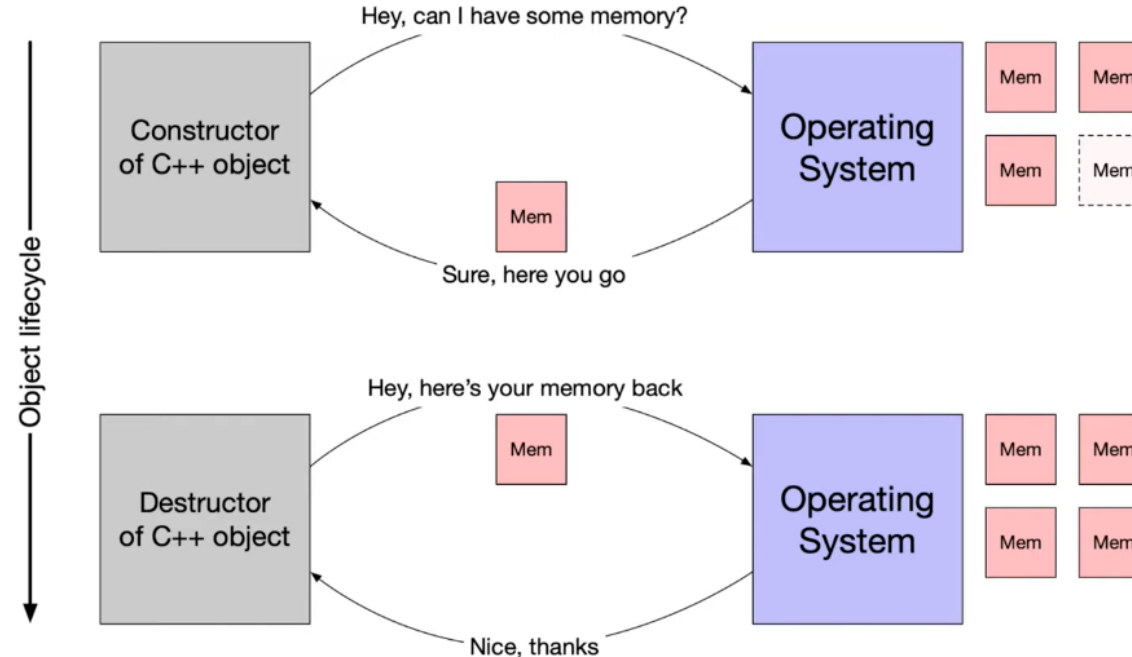
```
// problem #1's fix
{
    int *arr = new int[10];
    delete[] arr;
}
```

```
// problem #2's fix
{
    std::thread t1( [] () {
        // do some operations
    });
    t1.join();
}
```

```
// problem #3's fix
Std::mutex globalMutex;
Void func() {
    globalMutex.lock(); ....
    globalMutex.unlock();
}
```

# RAII (Resource Acquisition is initialization)

- RAII
  - When acquire resources in a constructor, also need to release them in the corresponding destructor
  - Resources:
    - Heap memory,
    - files,
    - sockets,
    - mutexes



# Locking

---std::mutex::lock(), unlock()

```
int    global_num = 0;
std::mutex    globalMutex;

void incre(int num){
    globalMutex.lock();
    global_num = global_num + 1;
    globalMutex.unlock();
}

int main(){
    std::thread threadA(incre, 10);
    std::thread threadB(incre, 10);
    threadA.join();
    threadB.join();
}
```

Is there a better ways to  
manage the mutex that  
can automatically unlock  
it when not used?



# Mutex and RAII locks



- `std::unique_lock`
- `std::scoped_lock`
- `std::shared_lock`

```
std::mutex my_mutex;  
{  
    std::unique_lock<std::mutex> lck(my_mutex);  
    ... ..  
}
```

```
{  
    std::unique_lock<std::mutex> lck(my_mutex);  
    ... ..  
}
```

```
std::shared_mutex shared_mutex;  
{  
    std::shared_lock<std::mutex> lck(shared_mutex);  
    ... ..  
}
```

# Locking

---unique\_lock

- A unique lock is an **object** that **manages a mutex object** with **unique ownership** in both states: locked and unlocked.
- RAll: When creating a local variable of type `std::unique_lock` passing the mutex as parameter.
  - On construction, the object **acquires a mutex object**, for whose locking and unlocking operations becomes responsible.
  - This class **guarantees** an **unlocked** status on **destruction** (even if not called explicitly).
- Features:
  - Deferred locking, Timeout locks, adoption of mutexes, movable(transfer of ownership)

# Locking

---unique\_lock

```
1  int    global_num = 0;
2  std::mutex    globalMutex;

3  void incre(int num){
4      std::unique_lock<std::mutex> u_lock(globalMutex);
5      global_num = global_num + 1;
6      ...
7  }

8  int main(){
9      std::thread t1(incre, 1);
10     std::thread t2(incre, 3);
11     t1.join();
12     t2.join();
    ...}
```

Only one  
thread could  
enter line 5-7  
at a time



# Locking

---unique\_lock

Unique\_lock feature: Deferred locking

```
std::mutex mtx;

void conditional_locking(bool should_lock) {
    // Create lock but do not acquire it
    std::unique_lock<std::mutex> lock(mtx, std::defer_lock);
    if (should_lock) {
        lock.lock();    // Conditionally acquire the lock
        std::cout << "Lock acquired." << std::endl;
    } else {
        std::cout << "Lock not acquired." << std::endl;
    }
}
```


```
int main() {
    std::thread t1(conditional_locking, true);
    std::thread t2(conditional_locking, false);
    t1.join();
    t2.join();
    return 0;
}
```

# Locking

---scoped\_lock

- Scoped\_lock: a mutex wrapper which obtains access to (locks) the provided mutex, and ensures it is unlocked when the scoped lock goes out of scope

```
1  int    global_num = 0;
2  std::mutex    globalMutex;
3
4  void incre(int num){
5      {
6          std::scoped_lock s_lock(globalMutex);
7          global_num = global_num + 1;
8      }
9      global_num = global_num + 1;
10     ...
11 }
12
```



# Locking

---shared\_lock

- `std::shared_lock` allows for shared ownership of mutexes.

```
std::shared_mutex mtx;
int global_val;
void print_val (int n, char c) {
    std::shared_lock<std::shared_mutex > lck (mtx);
    std::cout << global_val << std::endl;
}
int main () {
    std::thread th1 (print_val);
    std::thread th2 (print_val);
    th1.join();
    th2.join();
}
```

# RAII (Resource Acquisition is initialization)

*// problem #1*

*{*

*int \*arr = new int[10];*

*}*

*// arr goes out of scope but we didn't delete it, we now have a memory leak 😞*

*// problem #3*

*Std::mutex globalMutex;*

*Void func() {*

*globalMutex.lock();*

*}*

*// if we never unlocked the mutex(or exception occurred before unlock),  
it will cause a deadlock when other thread tries to acquire this lock 😞*



```
// problem #1's fix
```

```
{
```

```
    std::unique_ptr<int[]> arr(new int[10]);
```

```
.....
```

```
}
```

```
// problem #3's fix
```

```
Std::mutex globalMutex;
```

```
Void func() {
```

```
    std::unique_lock<std::mutex> lock(globalMutex);
```

```
....
```

```
}
```

# Locking

- `scoped_lock()`
- `unique_lock()`
- `shared_lock()`




## std::scoped\_lock

a mutex wrapper which obtains access to (locks) the provided mutex, and ensures it is unlocked when the scoped lock goes out of scope

When does s\_lock get released?

```
1  int          global_num = 0;
2  std::mutex    globalMutex;
3
4  void incre(int num){
5      {
6          std::scoped_lock s_lock(globalMutex);
7          global_num = global_num + 1;
8      }
9      global_num = global_num + 1;
10     ...
11 }
```



## std::scoped\_lock

```
std::vector<int> my_vec;
std::mutex      my_mutex;
void add_to_list(int new_value) {
    std::scoped_lock<std::mutex> lck(my_mutex);
    my_vec.push_back(new_value);
}
bool list_contains(int value_to_find) {
    std::scoped_lock<std::mutex> lck(my_mutex);
    return std::find(my_vec.begin(), my_vec.end(), value_to_find) != my_vec.end();
}
```



# Locking

- `scoped_lock()`
- `unique_lock()`
- `shared_lock()`

# std::unique\_lock

- A unique lock is an object that manages a mutex object with **unique ownership** in both states: locked and unlocked.
- RAll: When creating a local variable of type std::unique\_lock passing the mutex as parameter.
  - On construction, the object **acquires a mutex object**, for whose locking and unlocking operations becomes responsible.
  - This class **guarantees** an **unlocked** status on **destruction** (even if not called explicitly).
- Features:
  - Deferred locking, Timeout locks, adoption of mutexes, movable(transfer of ownership)

# Locking

- `scoped_lock()`
- `unique_lock()`
- `shared_lock()`

## std::shared\_lock

Shared\_lock allows for shared ownership of mutex. More than one thread could hold the mutex at the same time.

```
std::shared_mutex mtx;
int global_val;
void print_val (int n, char c) {
    std::shared_lock<std::shared_mutex > lck (mtx);
    std::cout << global_val << std::endl;
}
int main () {
    std::thread th1 (print_val);
    std::thread th2 (print_val);
    th1.join();
    th2.join();
    ... }
```

## Exercise

- How can I use the RAll class locks to implement R/W lock?
  - R/W locks allow multiple readers at the same time
  - But if there is writer, then there should be no readers, and only one writers.

# Where to find the resources?

- Concurrency programming:
  - [Book: C++ Concurrency in Action Practice Multithreading](#)
  - <https://learn.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe>
- Notes:
  - Atomic built-in: <https://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Atomic-Builtins.html>
  - Memory order: [https://cplusplus.com/reference/atomic/memory\\_order/#google\\_vignette](https://cplusplus.com/reference/atomic/memory_order/#google_vignette)