CS4414 Recitation 10

multi-threading II

11/01/2024

Alicia Yang

Winners of HW3 competition

Code that process .dat files in real-time:

1st place: Jeffrey Qian

2nd place: Arjun Saini

3rd place: Nam Anh Dang

4th place: Tianyi Zhang

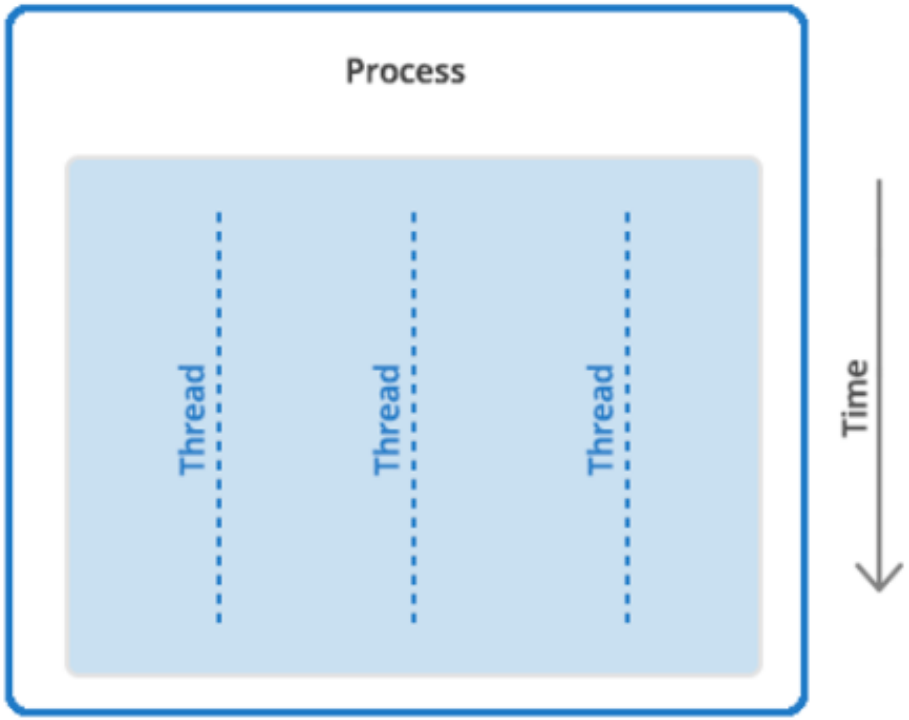
Special prices (code that uses cache.dat between runs)

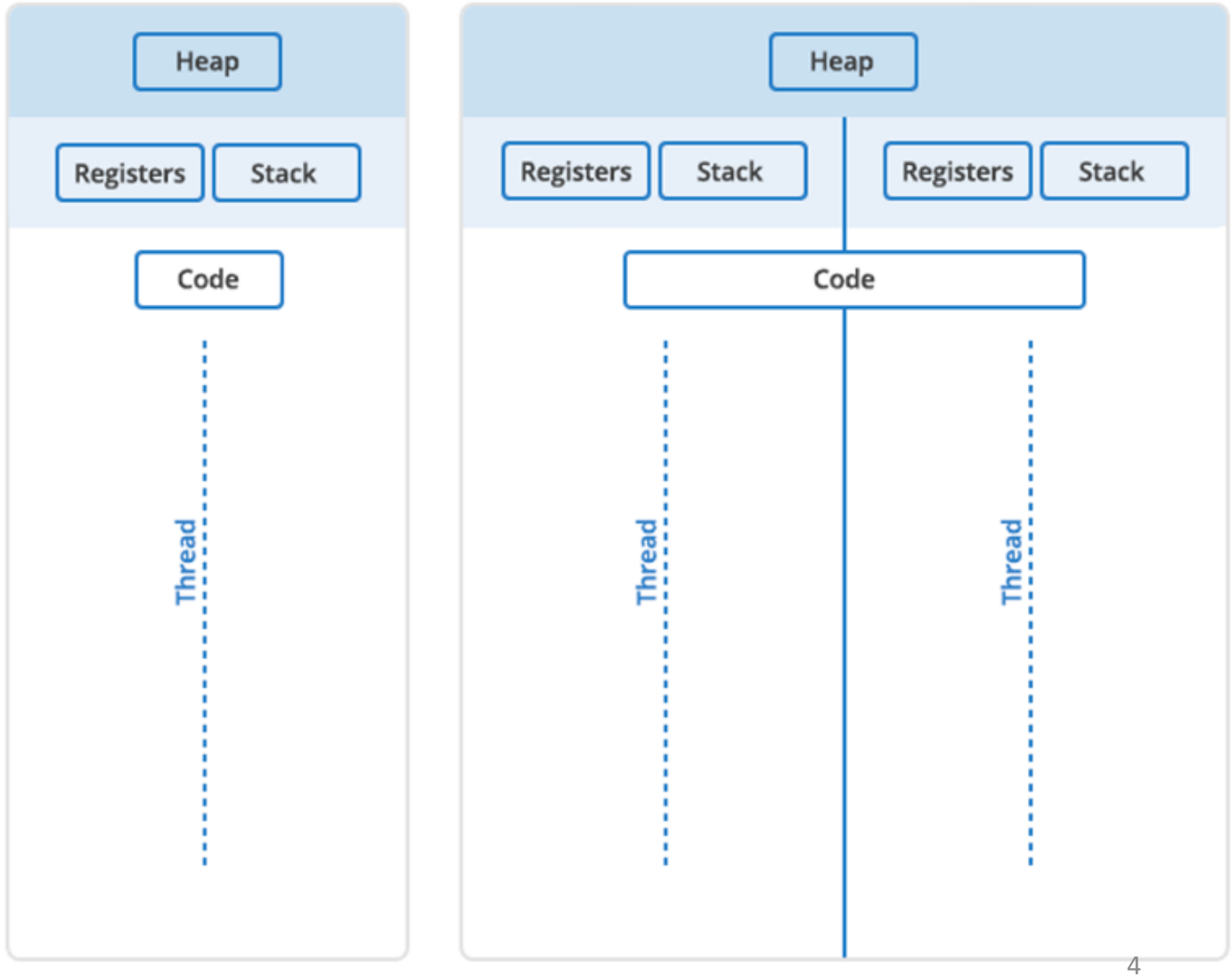
Peter Engel, Tami Takada, Reevu Adakroy

Recap

- Multithreading
- Race condition

Concurrency





https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/

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.

Code source:
https://github.com/aliciayuting/CS4414Demo.git

Sharing data among threads

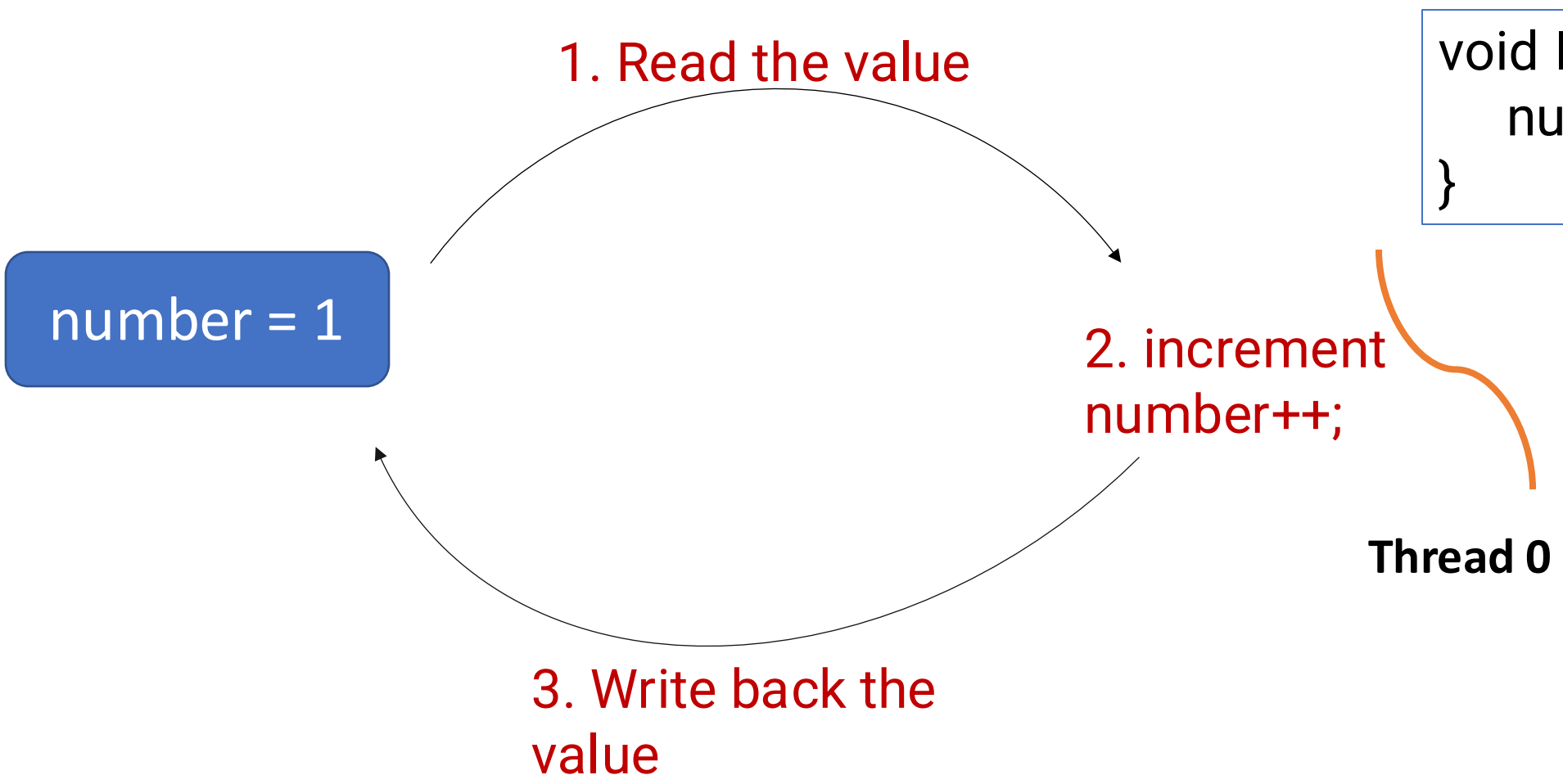
---race condition

- 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

Code source: https://github.com/aliciayuting/CS4414Demo.git

Example: Concurrent increments of a shared integer variable




```
void Increment(){
    number ++;
}
```

Example: Concurrent increments of a shared integer variable

num = 0

Read the value

1

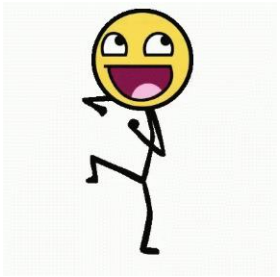
2

Write back the value

Oh! I see num=0.

Thread 1. Write back 1.

Ideally what we want



num = 3

3

4

Oh! I see num=1.

Thread 2. Write back 2.

5

6

Oh! I see num=2.

Thread 3. Write back 3.

Example: Concurrent increments of a shared integer variable

num = 0

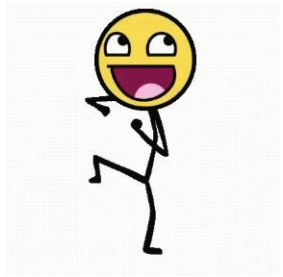
Read the value

Write back the value

Oh! I see num=0.

Thread 1. Write back 1.

Ideally what we want



num = 3

Will it always be in this sequence?

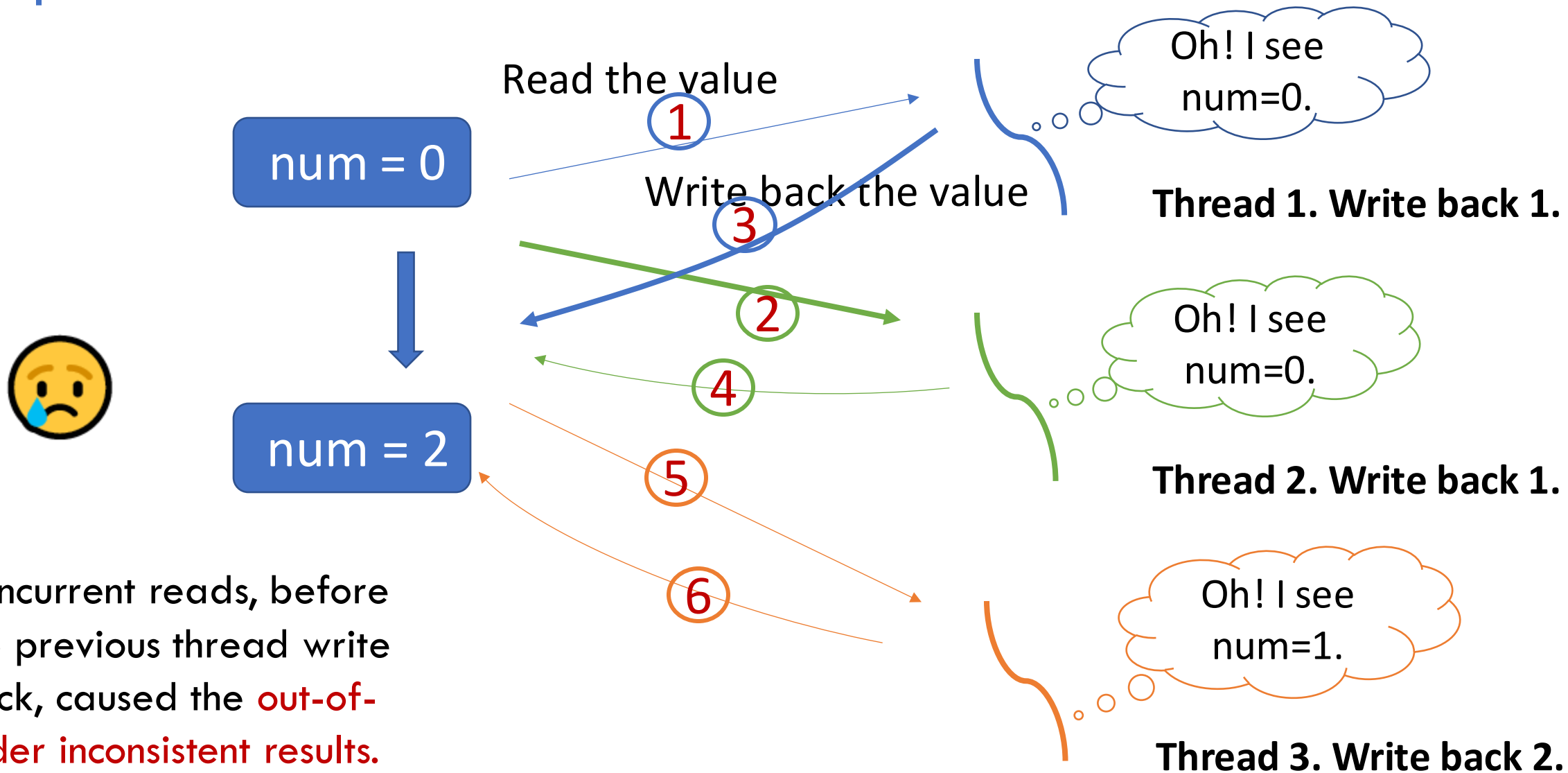
Oh! I see um=1.

ead 2. Write back 2.

Oh! I see num=2.

Thread 3. Write back 3.

Example: Concurrent increments of a shared integer variable



Concurrent reads, before the previous thread write back, caused the out-of-order inconsistent results.

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

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)

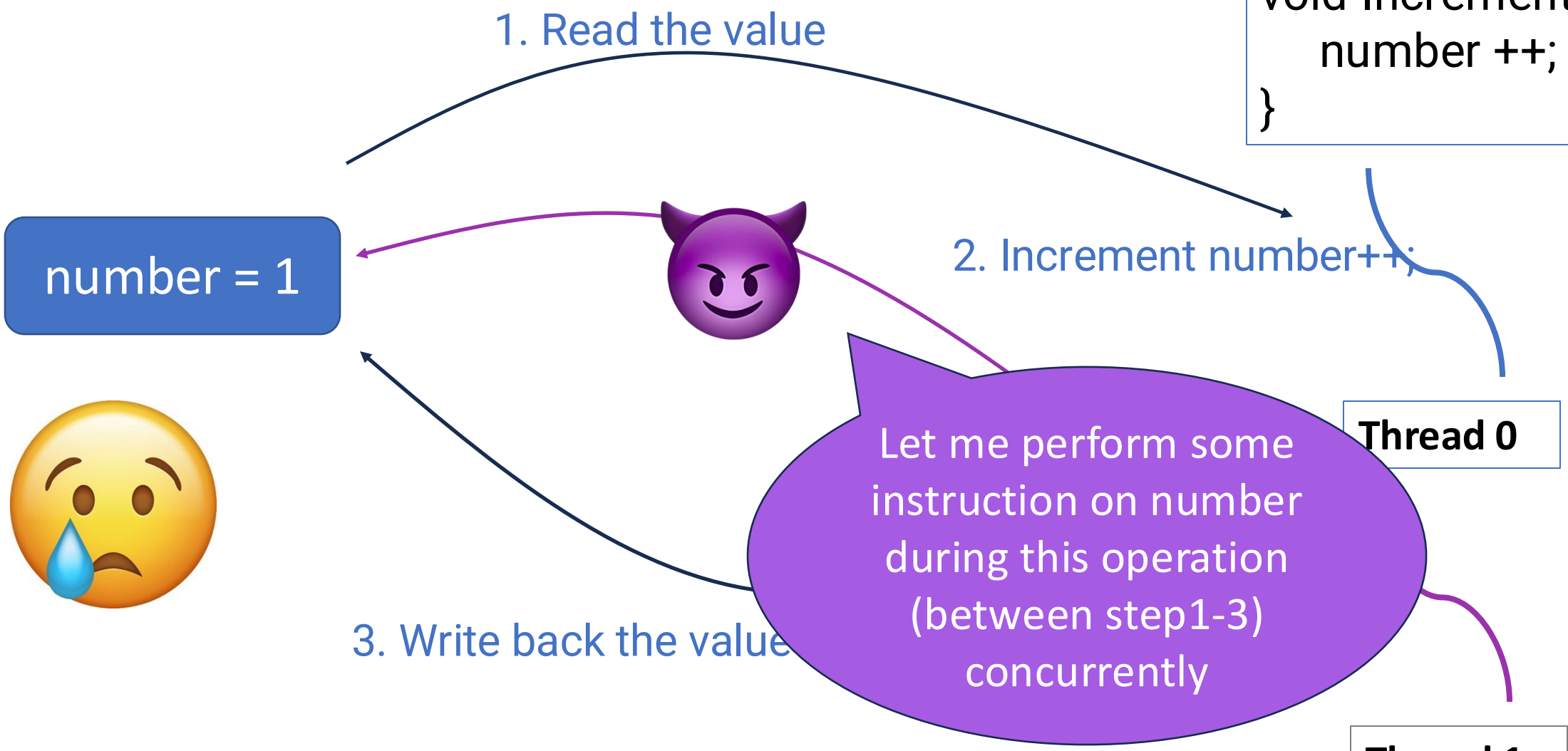
https://en.cppreference.com/w/cpp/atomic/atomic

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

```
void Increment(){
    number ++;
}
```



Data race condition: non-atomic access pattern

COMPILER EXPLORER Add... More Support diversity in C++ with #include <C++> × intel PC-lint Solid Sands Share Policies Other

C++ source #1 ×

(3, 14) C++

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

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

x86-64 gcc 11.2 Compiler options...

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

```
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

COMPILER EXPLORER Add... More Templates Sponsors JETBRAINS think-cell Share Policies

C++ source #1

Save/Load + Add new... Vim CppInsights Quick-bench C++

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

x86-64 gcc 14.2 (Editor #1)

x86-64 gcc 14.2 Compiler options...

Output... Filter... Libraries Overrides + Add new... Add tool...

```
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++(in
10          lea     rax, [rbp-4]
11          mov     rdi, rax
12          call    std::__atomic_base<int>::operator int()
13          leave
14          ret
```

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);`

https://en.cppreference.com/w/cpp/atomic/atomic

Memory_order

Accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by std::memory_order

- memory_order::relaxed

  // no synchronization or ordering constraints imposed on other reads or writes

- memory_order::consume

  // no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load

- memory_order::acquire

  // no reads or writes in the current thread can be reordered before this load.

- ….

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?

https://en.cppreference.com/w/cpp/atomic/atomic

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);
y ++;
```

What about y = y + 1?

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

https://en.cppreference.com/w/cpp/atomic/atomic

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

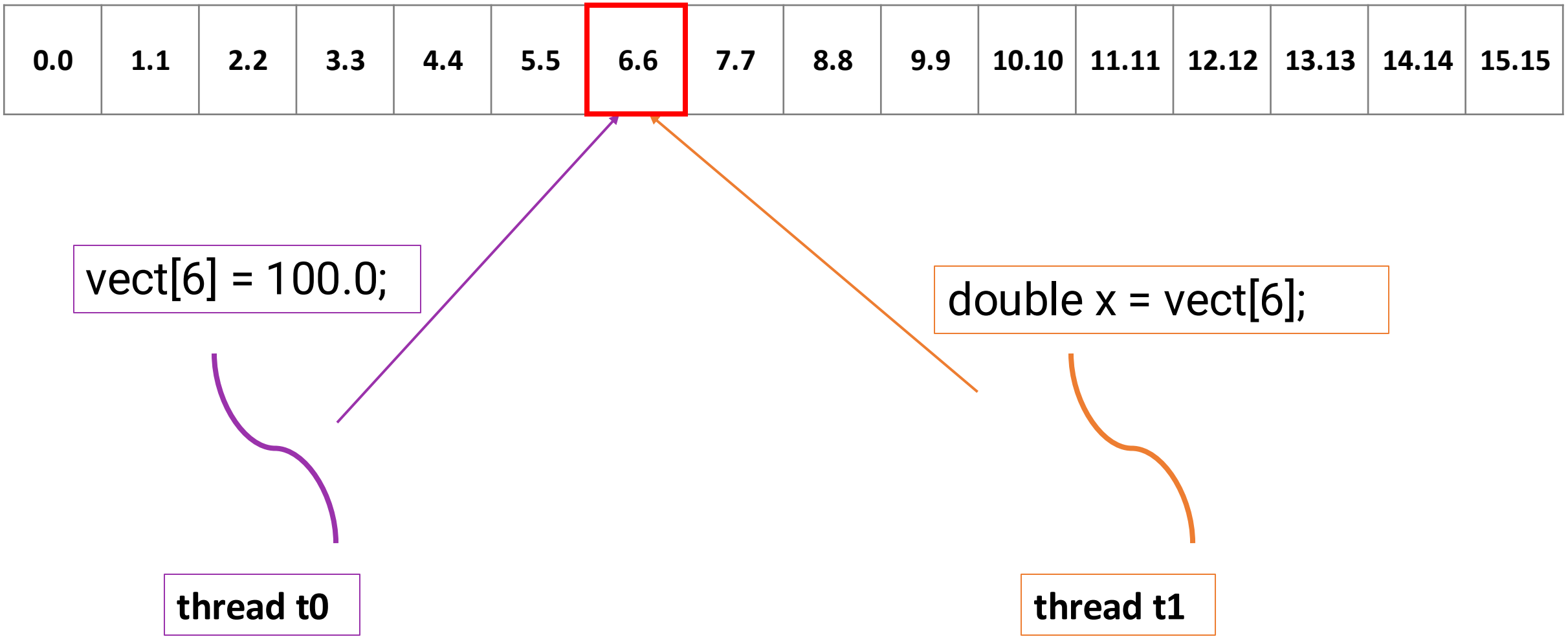
Read-only-access of std::vector

```
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

What if the threads are operating on different elements?

vect[6] = 100.0;

ect[6];

x could be 6.6 or 100.0 after this.

thread t1

Concurrent modification

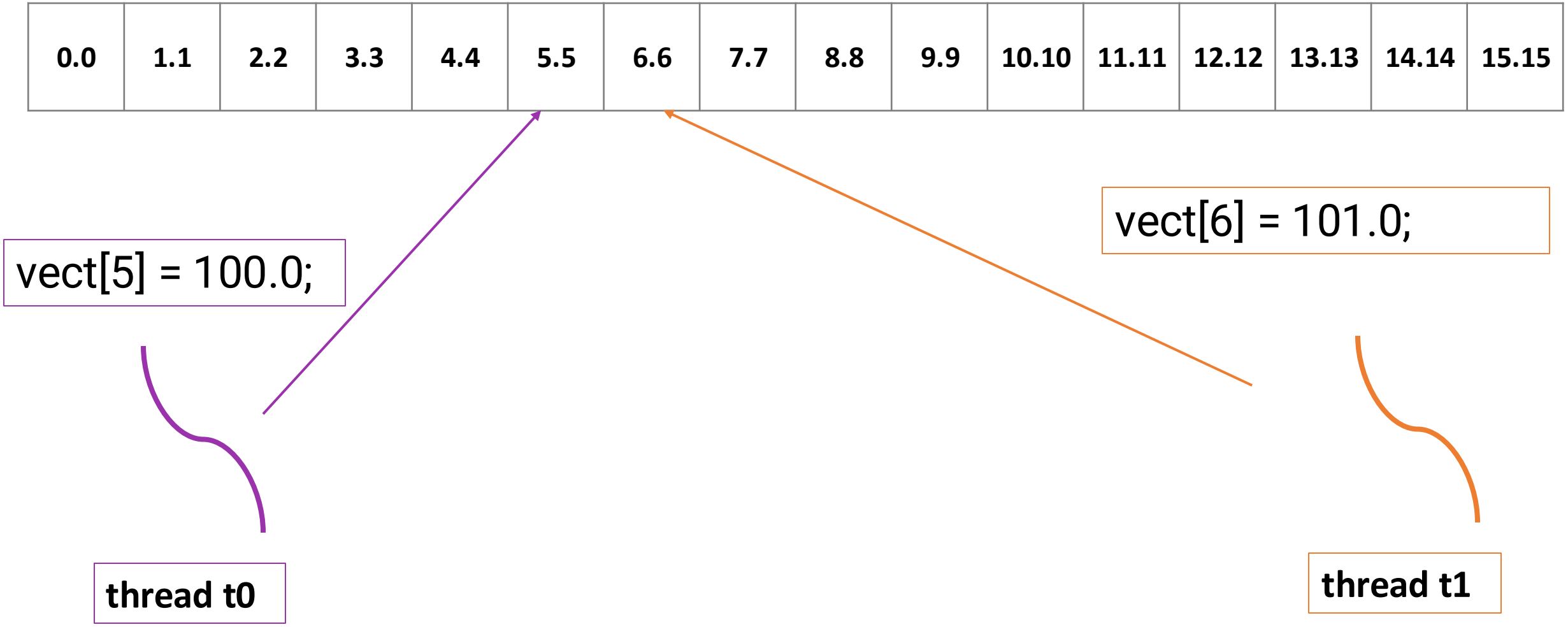
Is this code thread-safe?

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

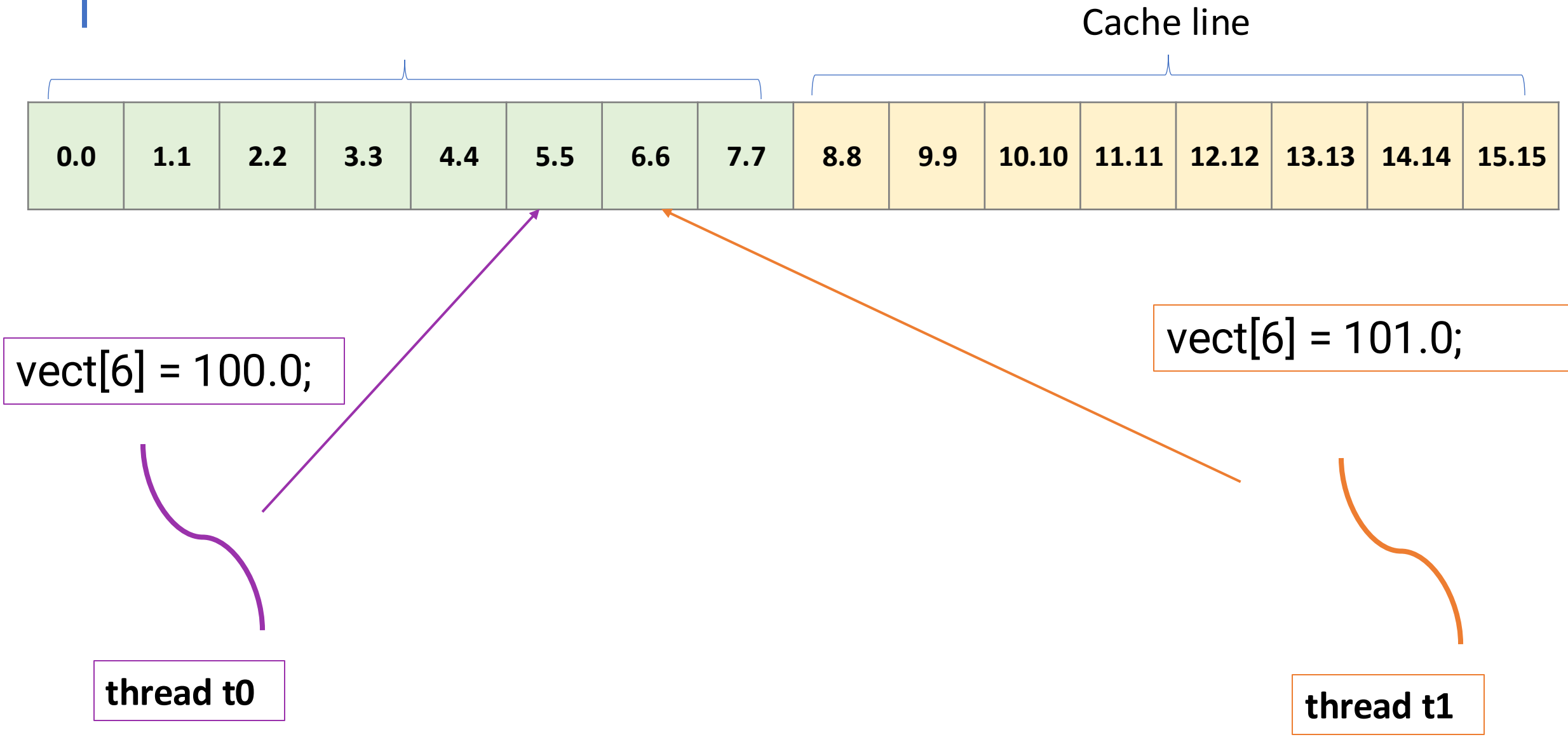
```
void set_value(std::vector<double>& vec, size_t index, double value) {
    vec[index] = value;
}
```

```
int main() {
    std::vector<double> vec(16) = {0.0, 1.1, 2.2, …. };
    std::thread t1(set_value, std::ref(vec), 5, 100.0);
    std::thread t2(set_value, std::ref(vec), 6, 101.0);
    t1.join();
    t2.join();
…}
```

Concurrent modification



Concurrent modification



Concurrent modification

At the risk of false sharing

Cache line

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;

vect[6] = 101.0;

thread t0

thread t1

It isn't thread-safe, due to false sharing. Each thread modifies a different element (vec[5] and vec[6]), but they may share the same cache line, and the modifications cause cache invalidations.

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);
    }
}
```

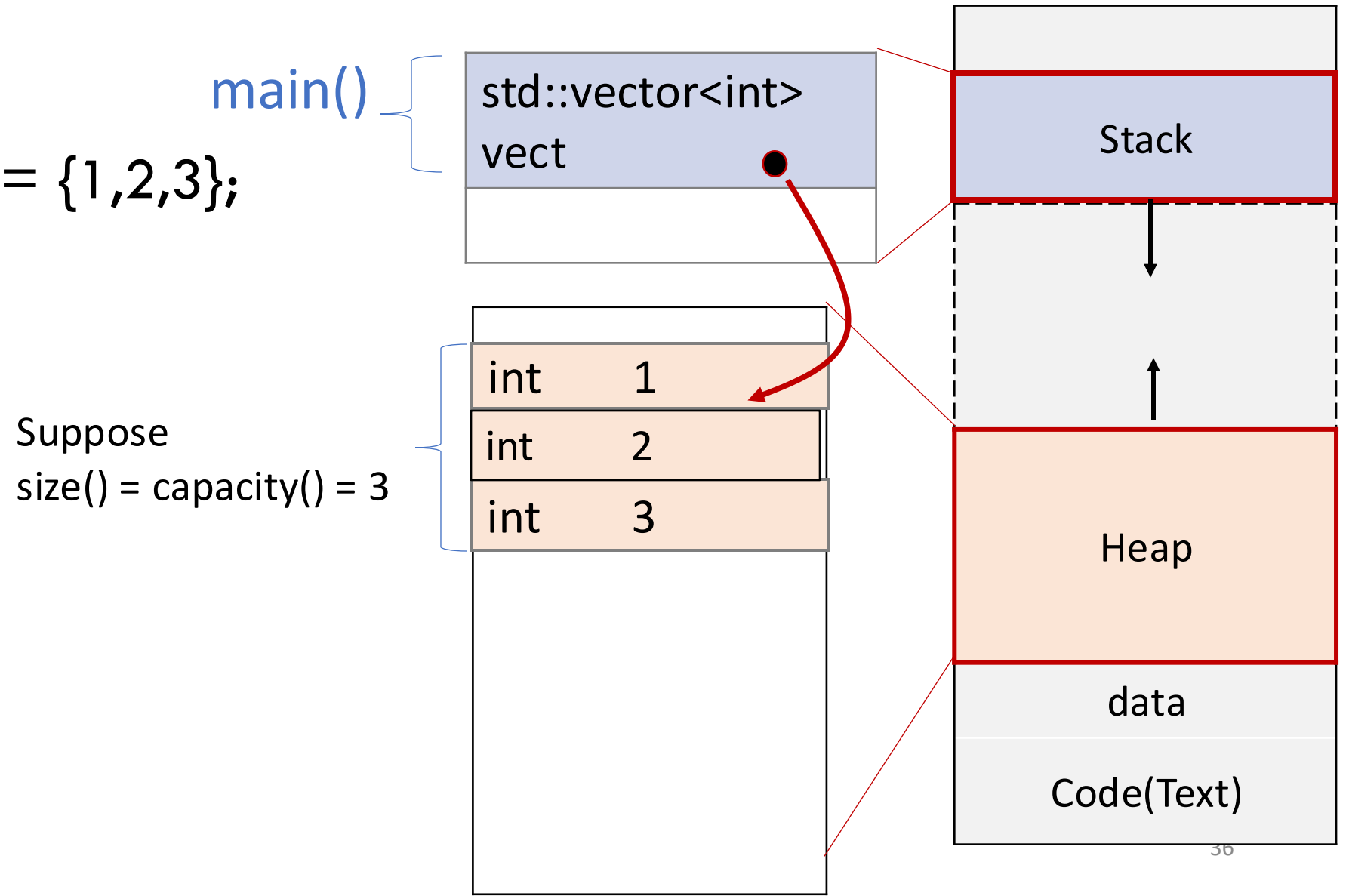
```
int main() {
   std::vector<int> vec = {1, 2, 3};
   std::thread t1(writer, std::ref(vec));
   std::thread t2([&vec]() {
      std::cout << "value: " << vec.back(); << std::endl;});

   t1.join();
   t2.join();
…}
```

How is std::vector allocated in memory

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


}
```

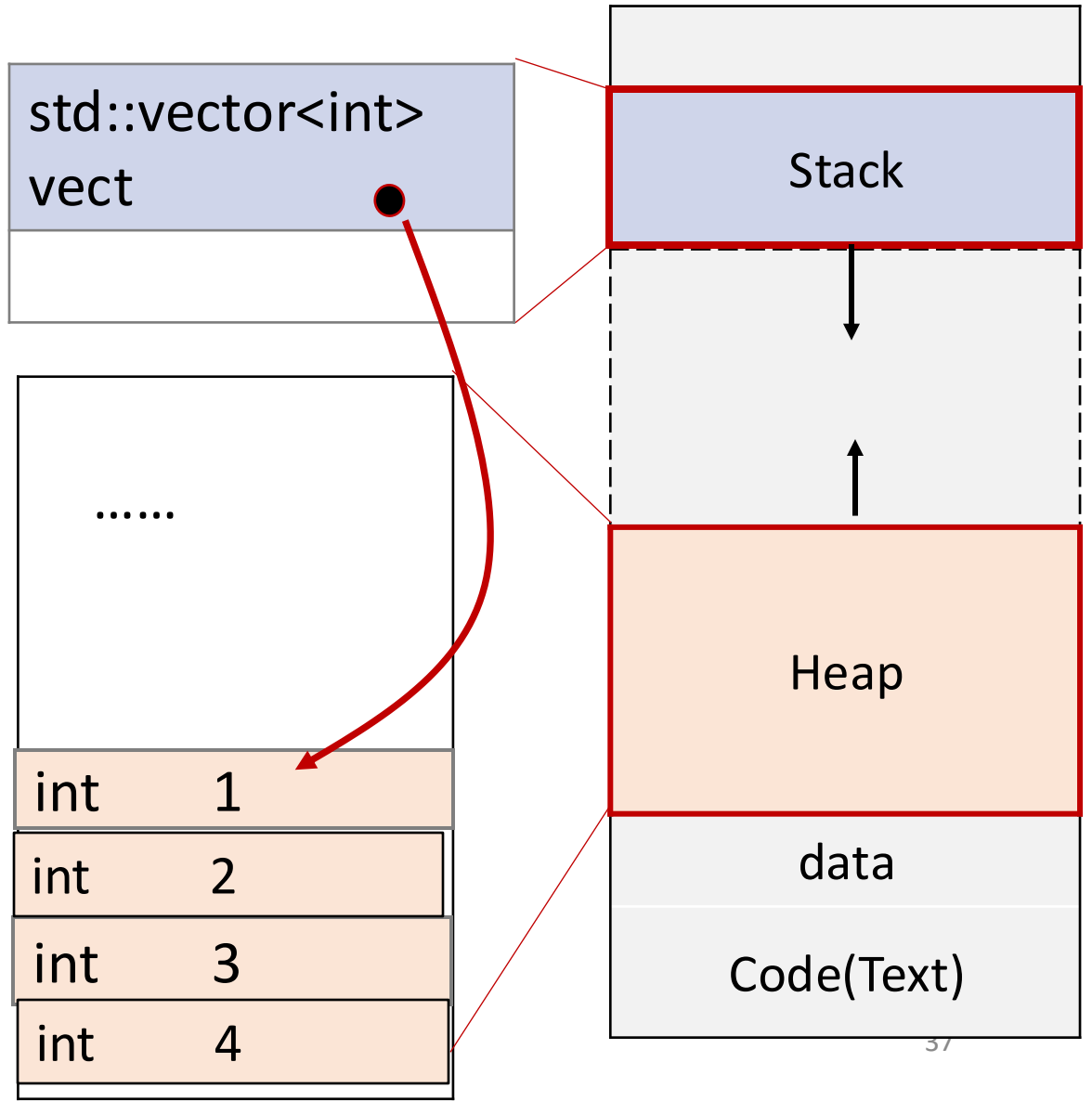


Suppose
size() = capacity() = 3

How is std::vector allocated in memory

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

main()



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.

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);
    }
}
```

```
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();
…}
```

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.

std::map

demo

```
std::map<int, int> global_map;

int main(){
      for (int i = 0; i < 1000000; ++i){
            global_map[i] = i;
      }
      std::thread r_thread(read_map);
      std::thread e_thread(erase_map);

      read_map_thread.join();
      erase_map_thread.join();
}
```

```
void read_map(){
     for (int i=0;i<1000000;++i){
          if(global_map.find(i) == global_map.end())
               continue;
          int val = global_map.at(i);
          if(val != i){
               std::cout << i << "," << val << std::endl;
          }
     }
}
```

```
void erase_map(){
      for (int i = 20000; i < 80000; ++i){
            global_map.erase(i);
      }
}
```

What could go wrong?

std::map

Multithread concurrent read and modification leads to race condition

```
std::map<int, int> global_map;

int main(){
      for (int i = 0; i < 1
            global_map[i] =
      }
      std::thread r_thread(i
      std::thread e_thread(erase

      r                .join();
      e                d.join();
}
```

What if I need multithreads to concurrently share such data?

```
void read_map(){
     for (int i=0;i<1000000;++i){
          if(global_map.find(i) == global_map.end())
               continue;
          al = global_map.at(i);
          = i){
          out << i << "," << val << std::endl;
     }
}
```

```
d erase_map(){
     for (int i = 20000; i < 80000; ++i){
          global_map.erase(i);
     }
}
```

Locking

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.
- RAII: 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 RAII 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 programing:
  - 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