High Performance Computing with C++

LSEG Technology

19th November 2022

LSEG

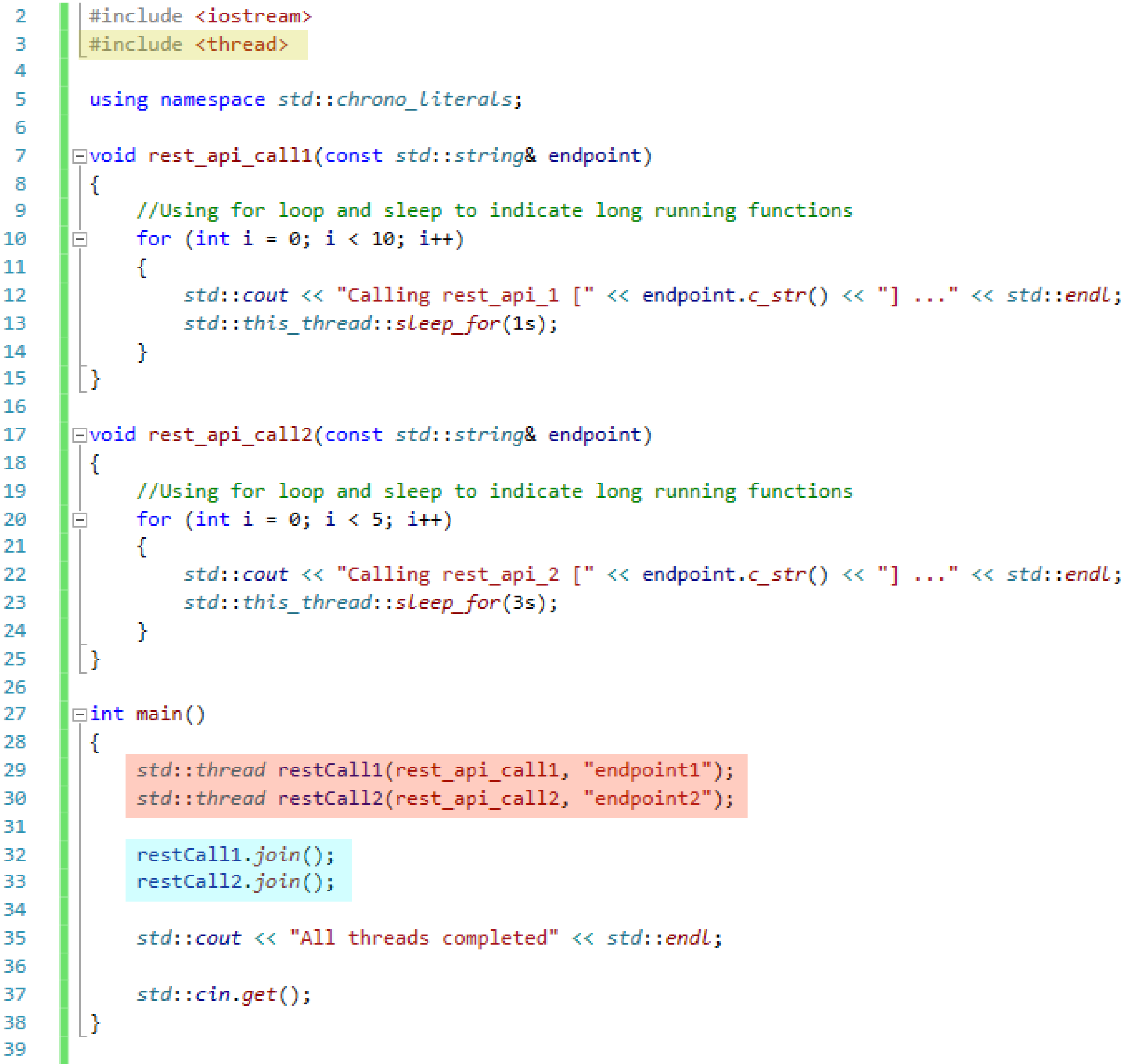
Multithreading in C++

- Threads allow multiple functions to execute concurrently
- Share same address space
- Address performance concerns
- Do tasks in parallel
- Before C++11, pthread. with C++11 std::thread

std::thread

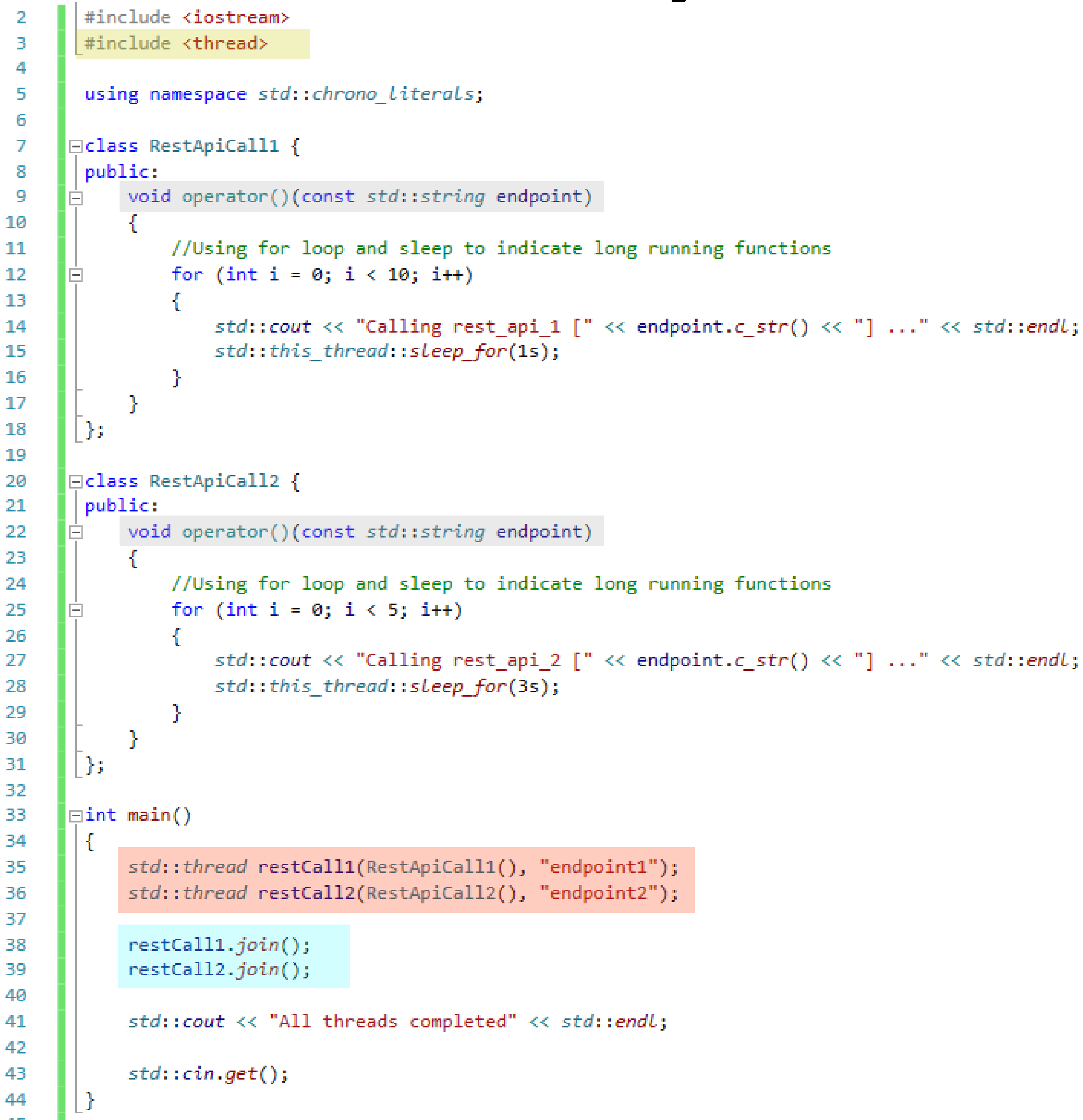
- Represents a single thread in C++
- new thread object with callable object
- A function pointer
- A function object
- A lambda expression

std::thread – Function pointers



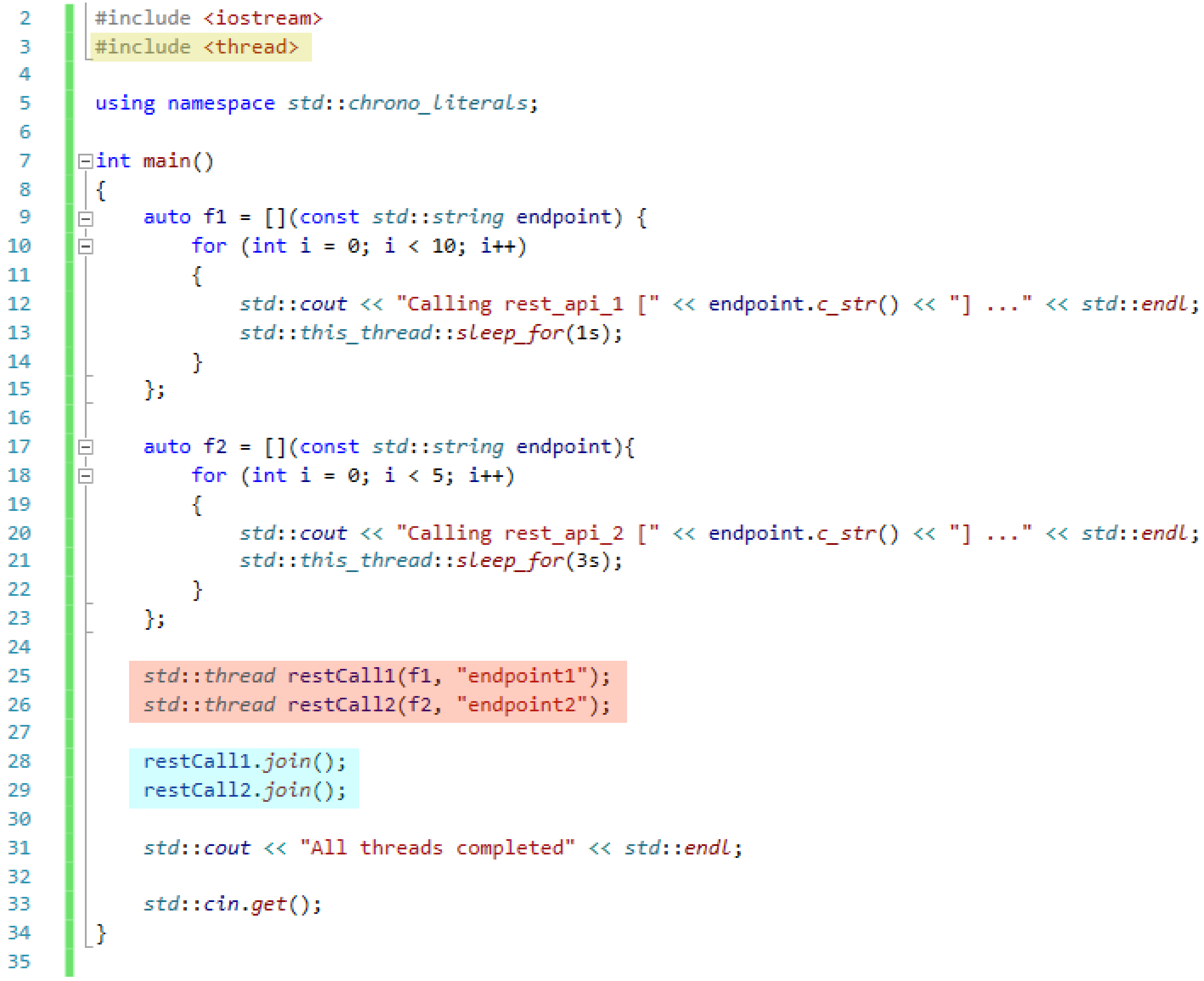
```cpp
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void rest_api_call1(const std::string& endpoint)
{
    //Using for loop and sleep to indicate long running functions
    for (int i = 0; i < 10; i++)
    {
        std::cout << "Calling rest_api_1 [" << endpoint.c_str() << "] ..." << std::endl;
        std::this_thread::sleep_for(1s);
    }
}

void rest_api_call2(const std::string& endpoint)
{
    //Using for loop and sleep to indicate long running functions
    for (int i = 0; i < 5; i++)
    {
        std::cout << "Calling rest_api_2 [" << endpoint.c_str() << "] ..." << std::endl;
        std::this_thread::sleep_for(3s);
    }
}

int main()
{
    std::thread restCall1(rest_api_call1, "endpoint1");
    std::thread restCall2(rest_api_call2, "endpoint2");

    restCall1.join();
    restCall2.join();

    std::cout << "All threads completed" << std::endl;

    std::cin.get();
}
```

std::thread – Function object



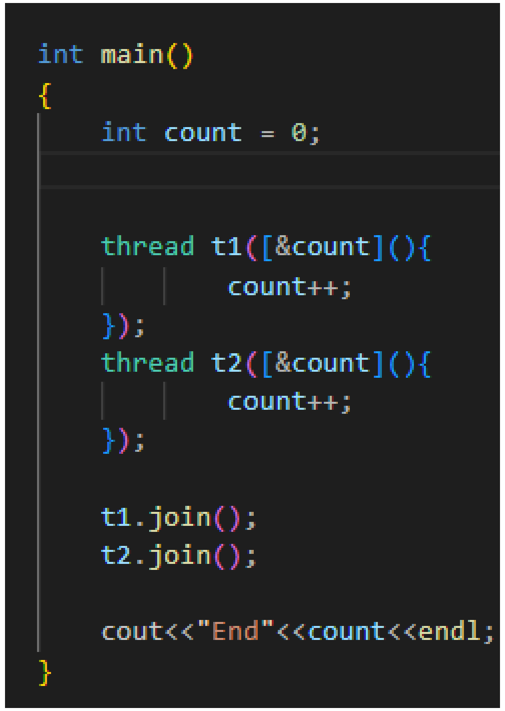
```cpp
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

class RestApiCall1 {
public:
    void operator()(const std::string endpoint)
    {
        //Using for loop and sleep to indicate long running functions
        for (int i = 0; i < 10; i++)
        {
            std::cout << "Calling rest_api_1 [" << endpoint.c_str() << "] ..." << std::endl;
            std::this_thread::sleep_for(1s);
        }
    }
};

class RestApiCall2 {
public:
    void operator()(const std::string endpoint)
    {
        //Using for loop and sleep to indicate long running functions
        for (int i = 0; i < 5; i++)
        {
            std::cout << "Calling rest_api_2 [" << endpoint.c_str() << "] ..." << std::endl;
            std::this_thread::sleep_for(3s);
        }
    }
};

int main()
{
    std::thread restCall1(RestApiCall1(), "endpoint1");
    std::thread restCall2(RestApiCall2(), "endpoint2");

    restCall1.join();
    restCall2.join();

    std::cout << "All threads completed" << std::endl;

    std::cin.get();
}
```

std::thread – Lambda expression

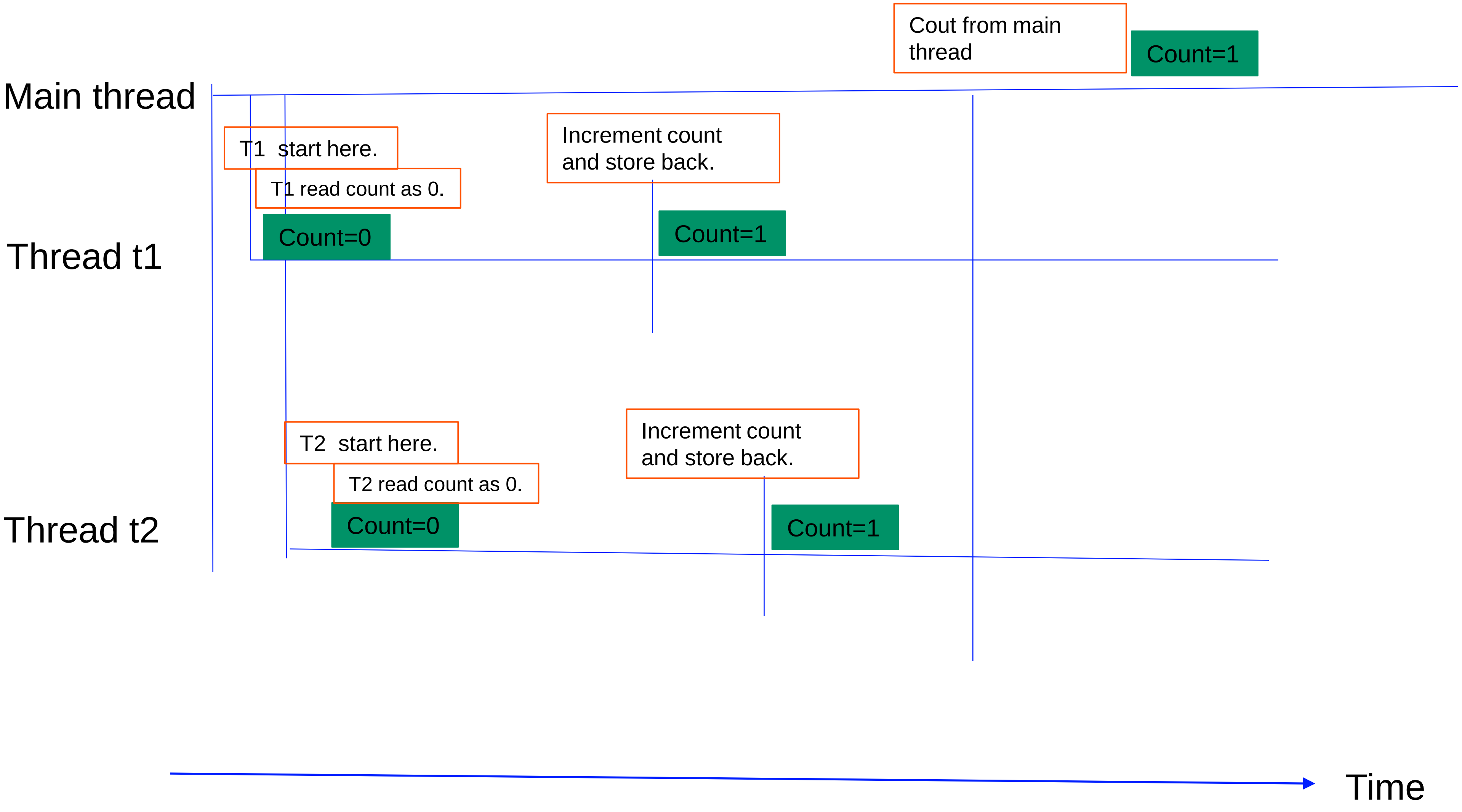


```cpp
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

int main()
{
    auto f1 = [](const std::string endpoint) {
        for (int i = 0; i < 10; i++)
        {
            std::cout << "Calling rest_api_1 [" << endpoint.c_str() << "] ..." << std::endl;
            std::this_thread::sleep_for(1s);
        }
    };

    auto f2 = [](const std::string endpoint){
        for (int i = 0; i < 5; i++)
        {
            std::cout << "Calling rest_api_2 [" << endpoint.c_str() << "] ..." << std::endl;
            std::this_thread::sleep_for(3s);
        }
    };

    std::thread restCall1(f1, "endpoint1");
    std::thread restCall2(f2, "endpoint2");

    restCall1.join();
    restCall2.join();

    std::cout << "All threads completed" << std::endl;

    std::cin.get();
}
```

Shared Data Between Threads



```
int main()
{
    int count = 0;


    thread t1([&count](){
            count++;
    });
    thread t2([&count](){
            count++;
    });

    t1.join();
    t2.join();

    cout<<"End"<<count<<endl;
}
```



Main thread

Thread t1

Thread t2

Cout from main thread

Count=1

T1 start here.

T1 read count as 0.

Count=0

Increment count and store back.

Count=1

T2 start here.

T2 read count as 0.

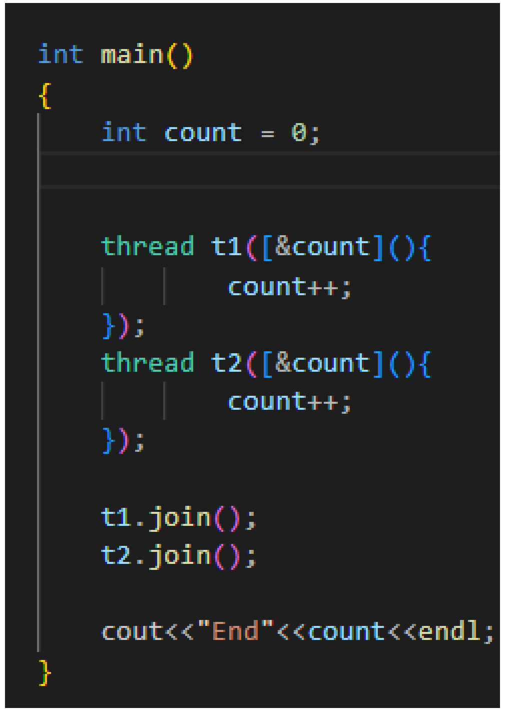
Count=0

Increment count and store back.

Count=1

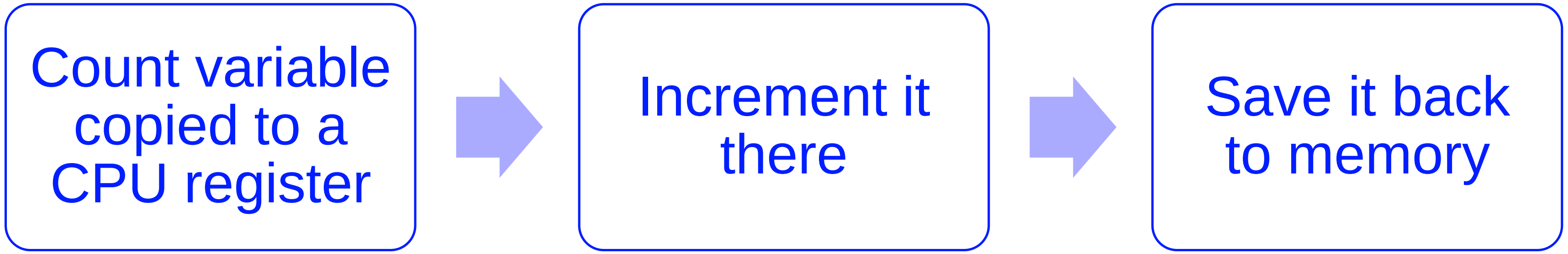
Time

What happens behind count ++



```
int main()
{
    int count = 0;


    thread t1([&count](){
            count++;
    });
    thread t2([&count](){
            count++;
    });

    t1.join();
    t2.join();

    cout<<"End"<<count<<endl;
}
```

Not an atomic operation!!

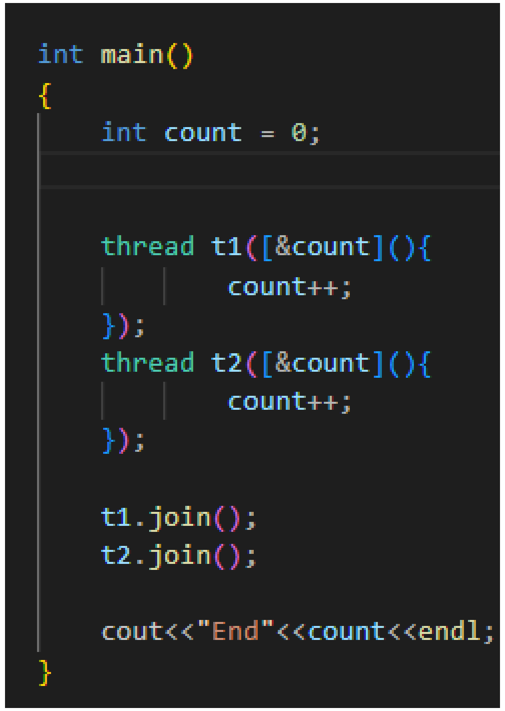


Count variable copied to a CPU register

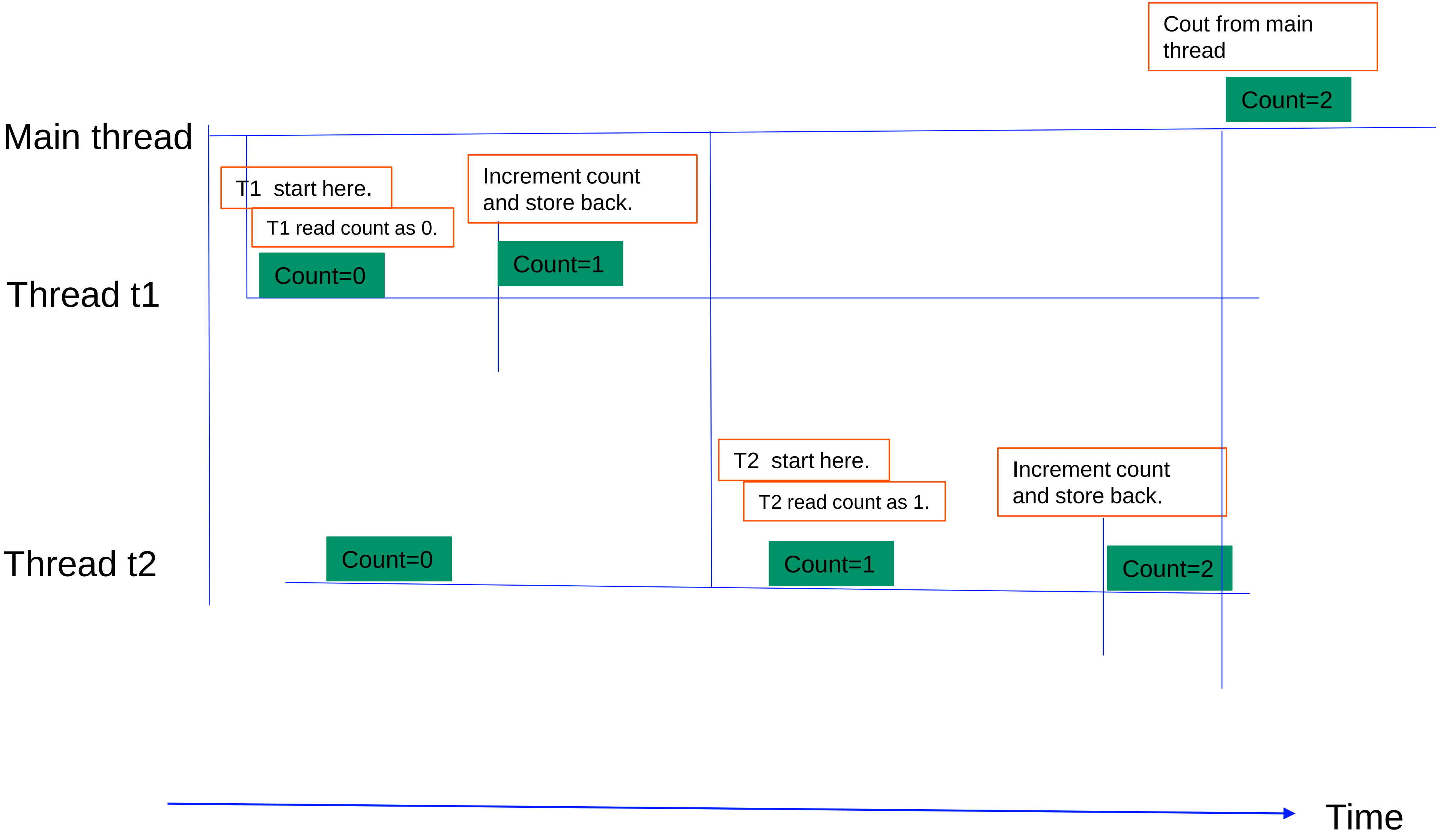
Increment it there

Save it back to memory

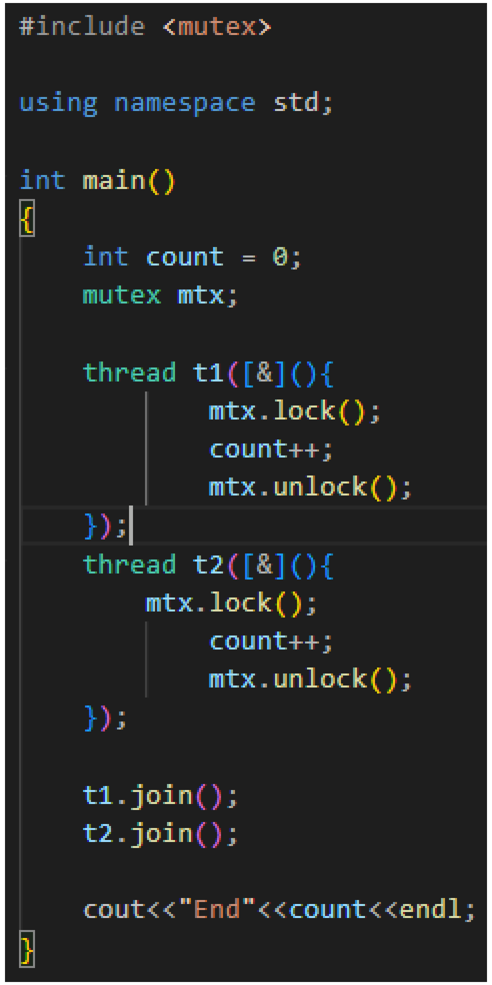
Shared Data Between Threads



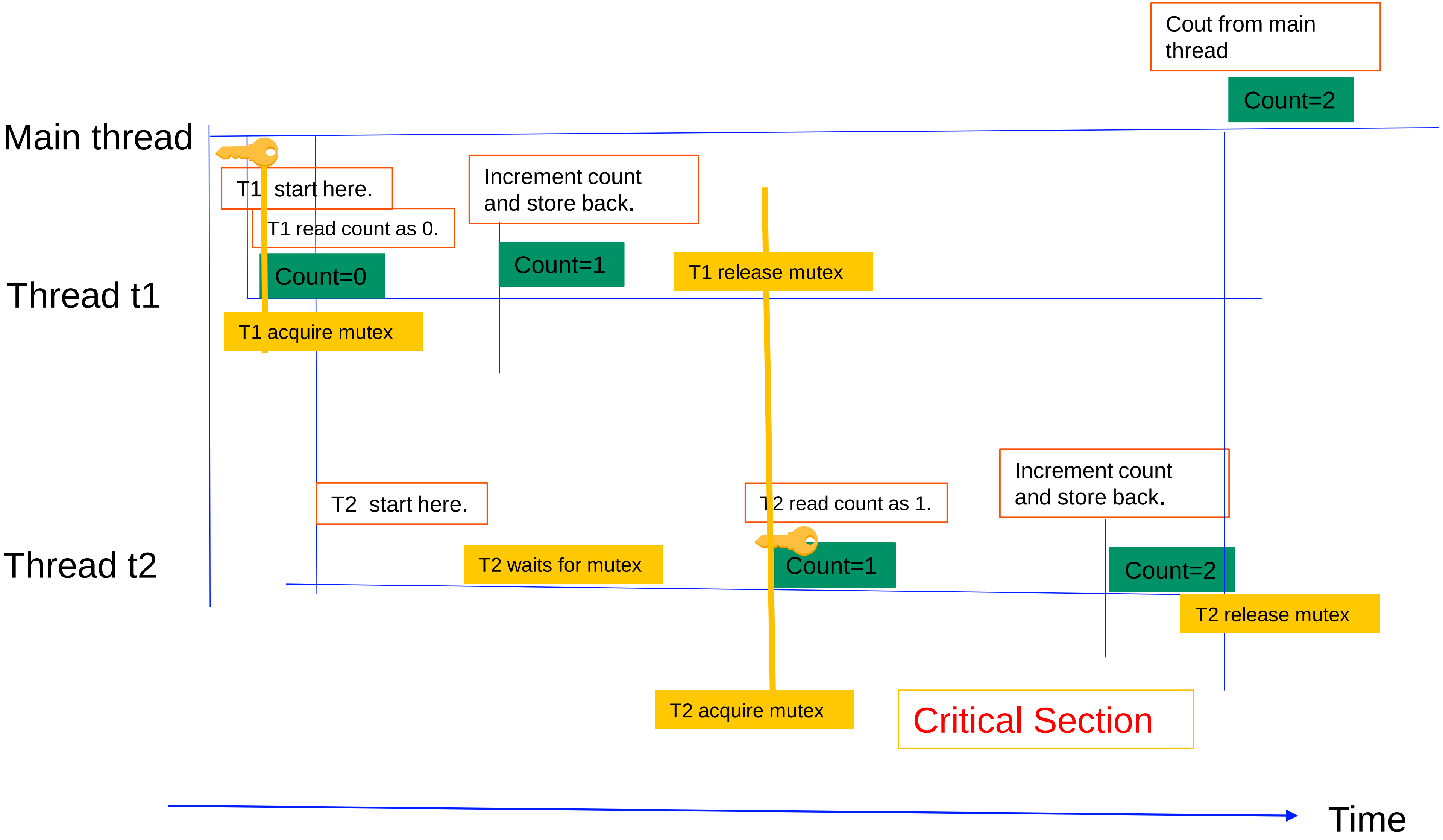
```
int main()
{
    int count = 0;


    thread t1([&count](){
            count++;
    });
    thread t2([&count](){
            count++;
    });

    t1.join();
    t2.join();

    cout<<"End"<<count<<endl;
}
```



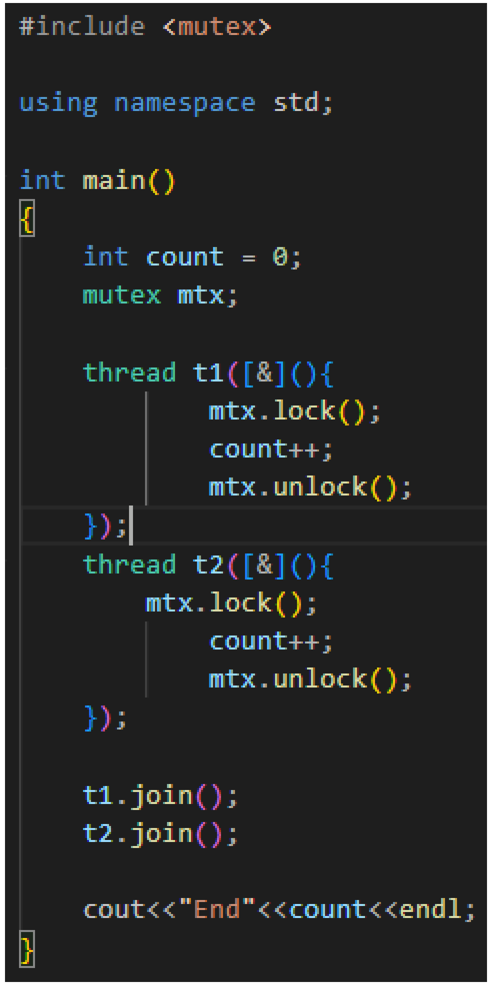
Mutex – a key to access shared data



```
#include <mutex>

using namespace std;

int main()
{
    int count = 0;
    mutex mtx;

    thread t1([&](){
            mtx.lock();
            count++;
            mtx.unlock();
    });
    thread t2([&](){
        mtx.lock();
            count++;
            mtx.unlock();
    });

    t1.join();
    t2.join();

    cout<<"End"<<count<<endl;
}
```



Mutex



```
#include <mutex>

using namespace std;

int main()
{
    int count = 0;
    mutex mtx;

    thread t1([&](){
            mtx.lock();
            count++;
            mtx.unlock();
    });
    thread t2([&](){
        mtx.lock();
            count++;
            mtx.unlock();
    });

    t1.join();
    t2.join();

    cout<<"End"<<count<<endl;
}
```

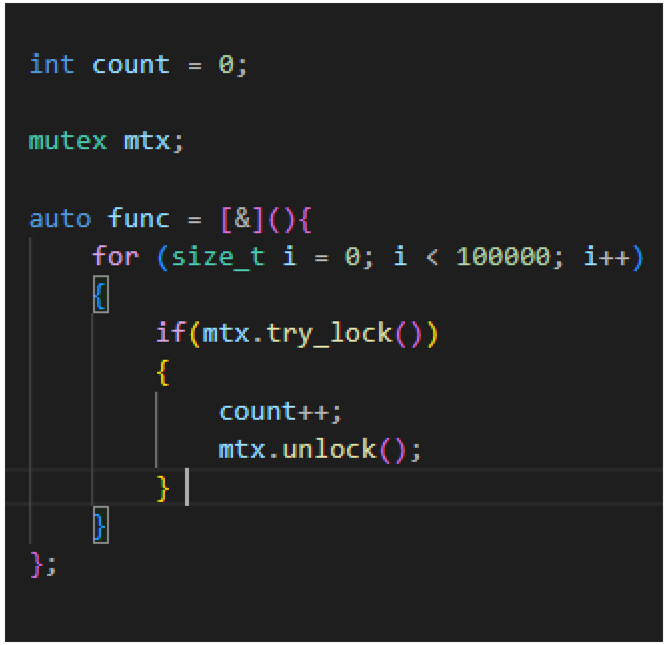
#include <mutex>

To protect shared data from being simultaneously accessed by multiple threads.

Need to make sure we unlock()

https://en.cppreference.com/w/cpp/thread/mutex

Mutex : try_lock()



```
int count = 0;

mutex mtx;

auto func = [&](){
    for (size_t i = 0; i < 100000; i++)
    {
        if(mtx.try_lock())
        {
            count++;
            mtx.unlock();
        }
    }
};
```

#include <mutex>

Try to get the lock

- ➢ If it gets lock: return back with true
- ➢ If it failed to get lock: return back with false

Anyway, returns back immediately !

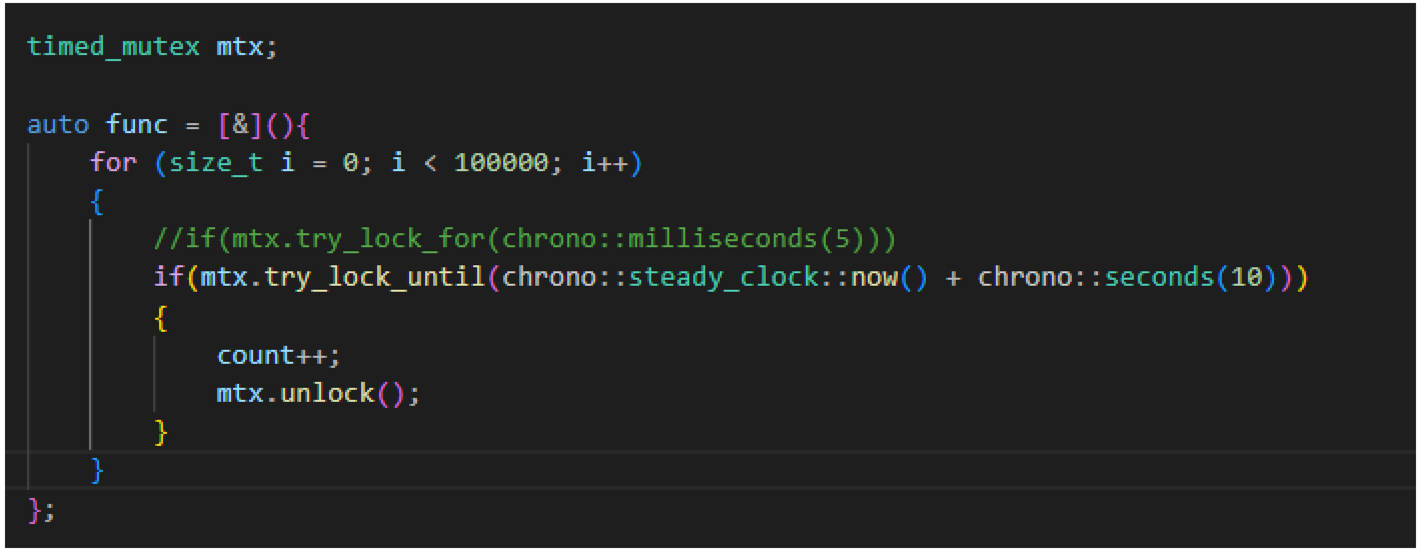
If lock acquired, need to call unlock() to release

May return false even if it is not owned by any other thread.

If try_lock is called by a thread that already owns the mutex, the behavior is undefined.

https://en.cppreference.com/w/cpp/thread/mutex/try_lock

Timed Mutex : try_lock_for() or try_lock_until



```
timed_mutex mtx;

auto func = [&](){
    for (size_t i = 0; i < 100000; i++)
    {
        //if(mtx.try_lock_for(chrono::milliseconds(5)))
        if(mtx.try_lock_until(chrono::steady_clock::now() + chrono::seconds(10)))
        {
            count++;
            mtx.unlock();
        }
    }
};
```

#include <mutex>

Similar to mutex

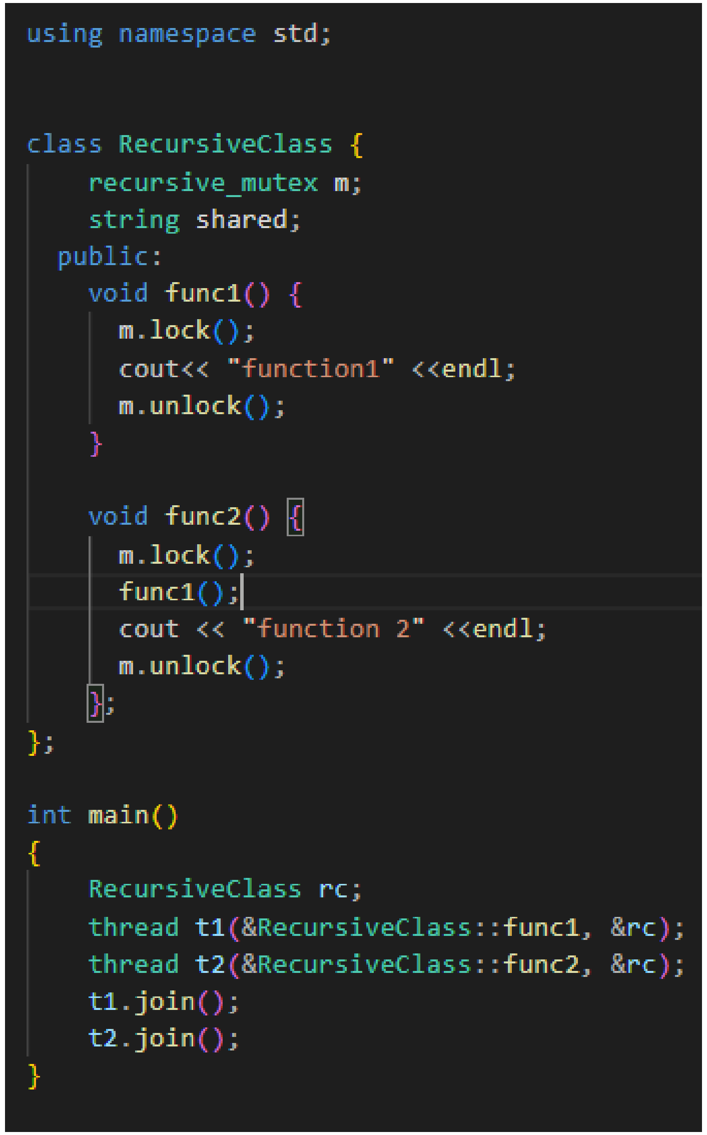
But can try to get the lock for a given time

- If it gets lock: return back with true
- If it failed to get lock: return back with false

Try_lock_for : for a given time duration, eg: for 5s
Try_lock_until : until a specified time point, eg: 11:50:50 th second

https://en.cppreference.com/w/cpp/thread/timed_mutex

Recursive Mutex



```cpp
using namespace std;


class RecursiveClass {
    recursive_mutex m;
    string shared;
  public:
    void func1() {
      m.lock();
      cout<< "function1" <<endl;
      m.unlock();
    }

    void func2() {
      m.lock();
      func1();
      cout << "function 2" <<endl;
      m.unlock();
    };
};

int main()
{
    RecursiveClass rc;
    thread t1(&RecursiveClass::func1, &rc);
    thread t2(&RecursiveClass::func2, &rc);
    t1.join();
    t2.join();
}
```

#include <mutex>

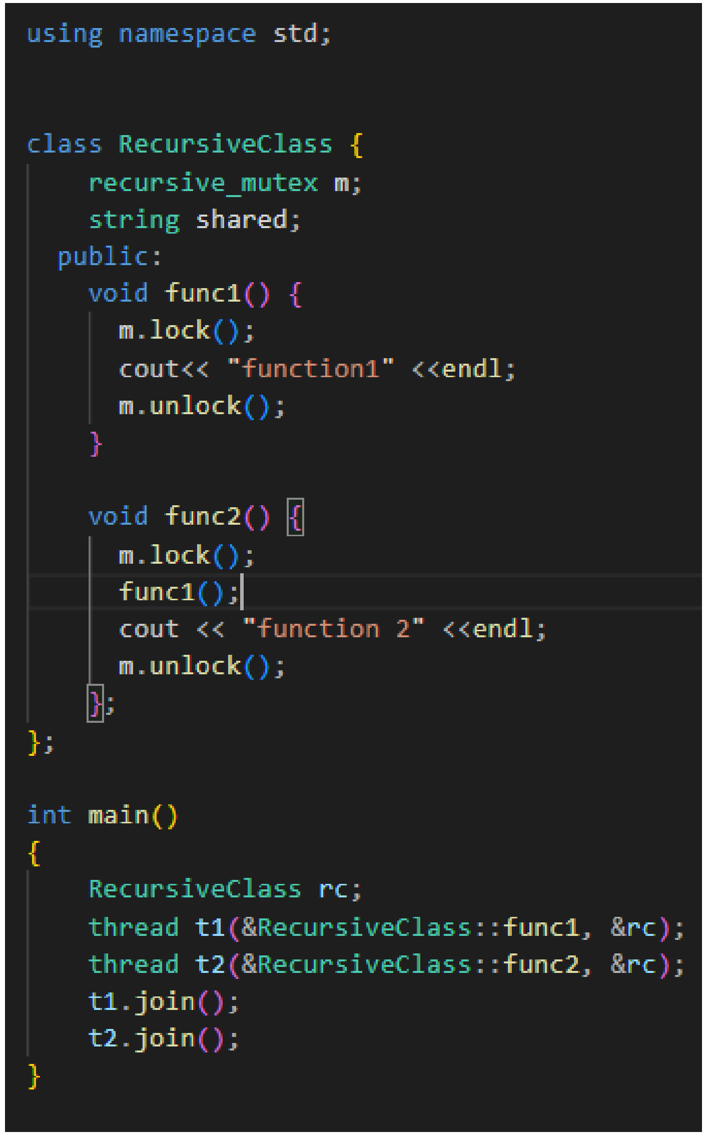
Similar to mutex + Provide recursive ownership

The thread which currently owns lock can call lock() recursively.

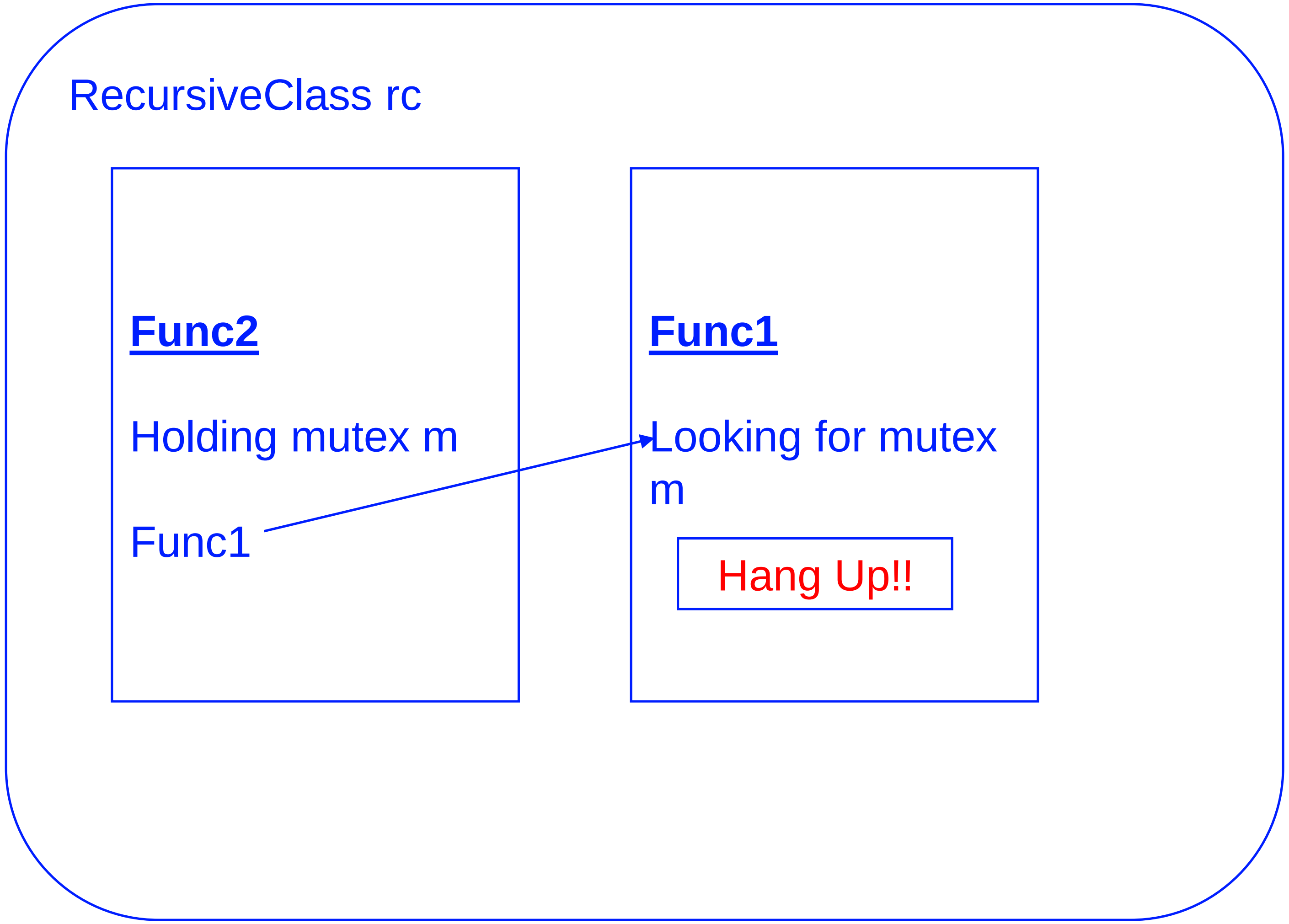
See recursive_timed_mutex too

https://en.cppreference.com/w/cpp/thread/recursive_mutex

Recursive Mutex



```
using namespace std;


class RecursiveClass {
    recursive_mutex m;
    string shared;
  public:
    void func1() {
      m.lock();
      cout<< "function1" <<endl;
      m.unlock();
    }

    void func2() {
      m.lock();
      func1();
      cout << "function 2" <<endl;
      m.unlock();
    };
};

int main()
{
    RecursiveClass rc;
    thread t1(&RecursiveClass::func1, &rc);
    thread t2(&RecursiveClass::func2, &rc);
    t1.join();
    t2.join();
}
```



RecursiveClass rc

Func2

Holding mutex m

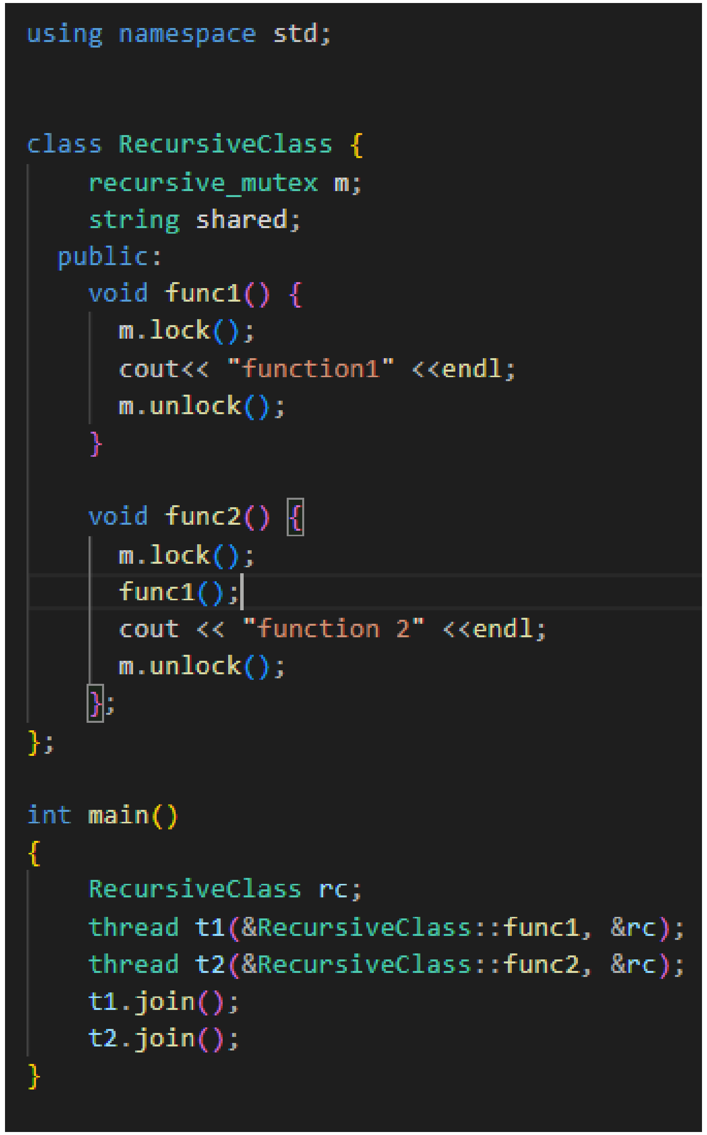
Func1

Func1

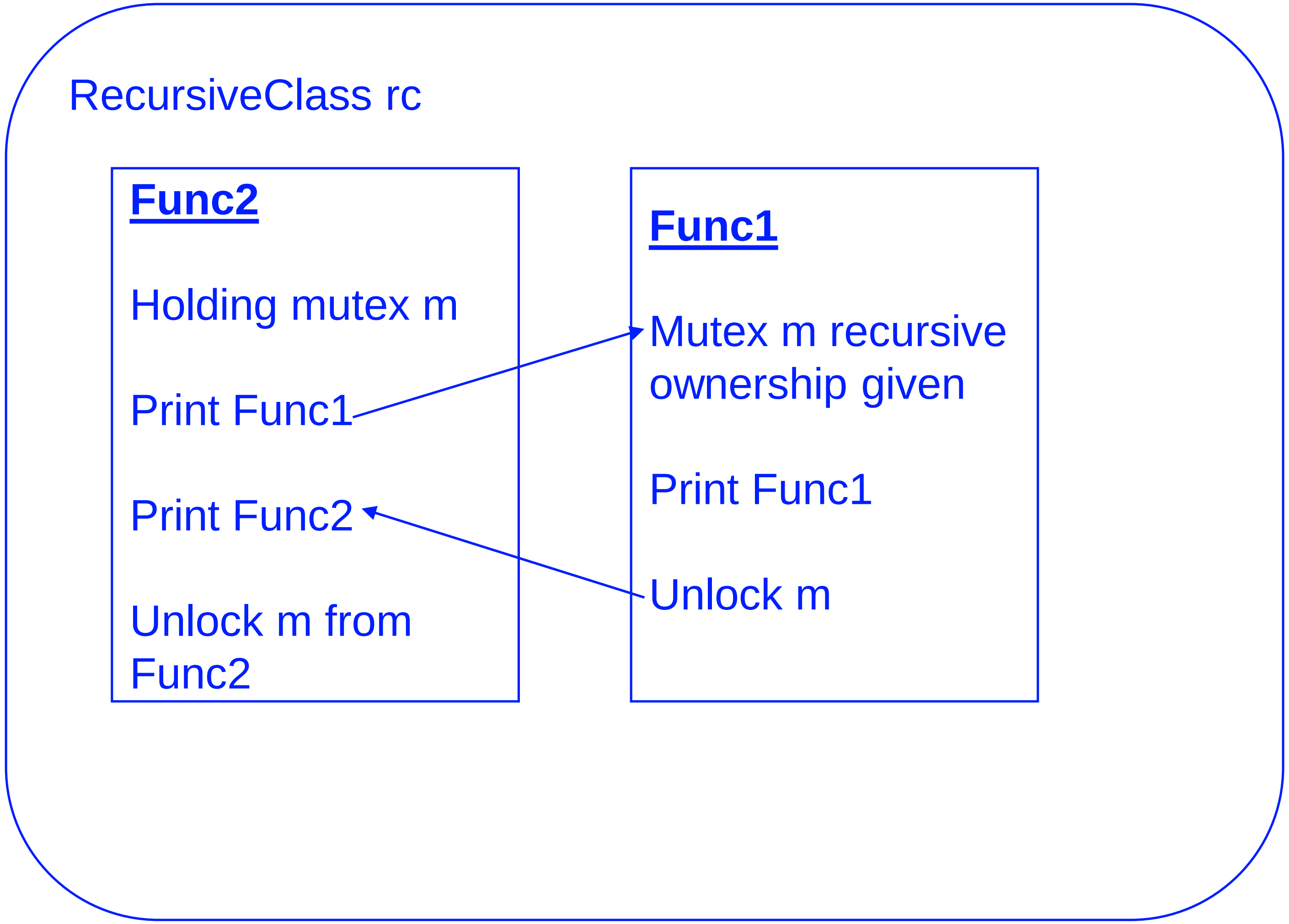
Looking for mutex m

Hang Up!!

Recursive Mutex



```
using namespace std;


class RecursiveClass {
    recursive_mutex m;
    string shared;
  public:
    void func1() {
      m.lock();
      cout<< "function1" <<endl;
      m.unlock();
    }

    void func2() {
      m.lock();
      func1();
      cout << "function 2" <<endl;
      m.unlock();
    };
};

int main()
{
    RecursiveClass rc;
    thread t1(&RecursiveClass::func1, &rc);
    thread t2(&RecursiveClass::func2, &rc);
    t1.join();
    t2.join();
}
```



RecursiveClass rc

Func2

Holding mutex m

Print Func1

Print Func2

Unlock m from Func2

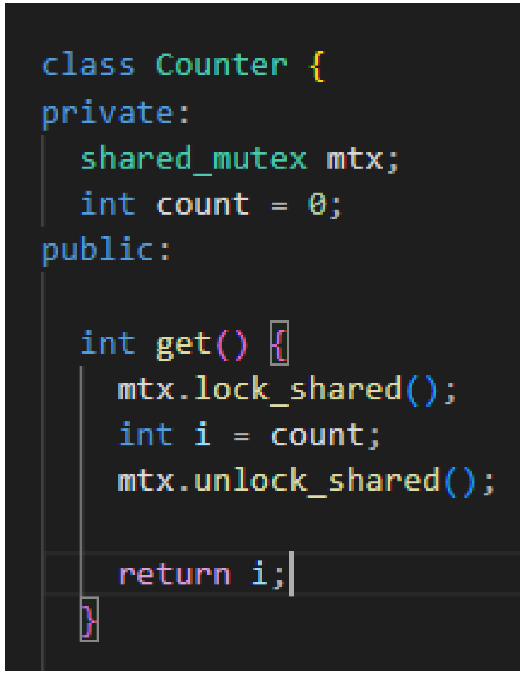
Func1

Mutex m recursive ownership given

Print Func1

Unlock m

Shared Mutex (C++17)



```
class Counter {
private:
  shared_mutex mtx;
  int count = 0;
public:

  int get() {
    mtx.lock_shared();
    int i = count;
    mtx.unlock_shared();

    return i;
  }
```

#include <shared_mutex>

Similar to mutex + Provide shared access

Read Write Lock Behavior

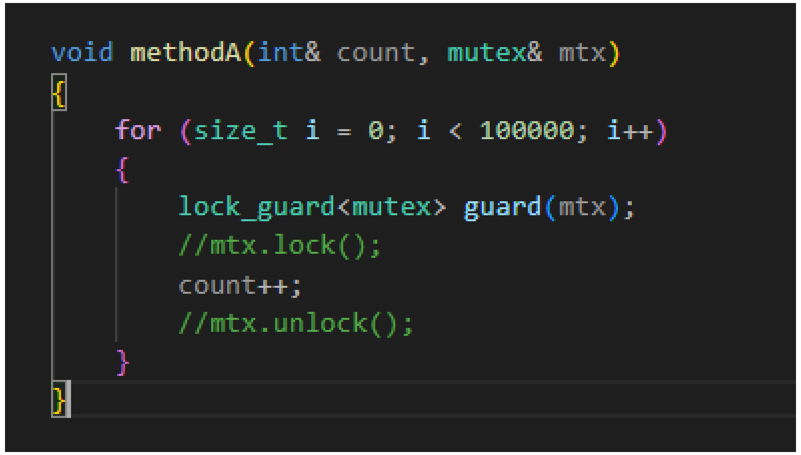
2 Levels of access:

- shared: several threads can share ownership
  - Possible if no other threads have taken an exclusive lock
  - lock_shared(), try_lock_shared(), unlock_shared
- exclusive: only 1 thread can own the mutex
  - lock(), try_lock(), unlock()

If 1 thread has taken a shared lock, other threads too can take shared lock. But, not exclusive lock

https://en.cppreference.com/w/cpp/thread/shared_mutex

Lock_guard



```
void methodA(int& count, mutex& mtx)
{
    for (size_t i = 0; i < 100000; i++)
    {
        lock_guard<mutex> guard(mtx);
        //mtx.lock();
        count++;
        //mtx.unlock();
    }
}
```

#include <mutex>

RAII mechanism : Resource Acquisition is Initialization

Light weight wrapper around mutex which make sure RAII

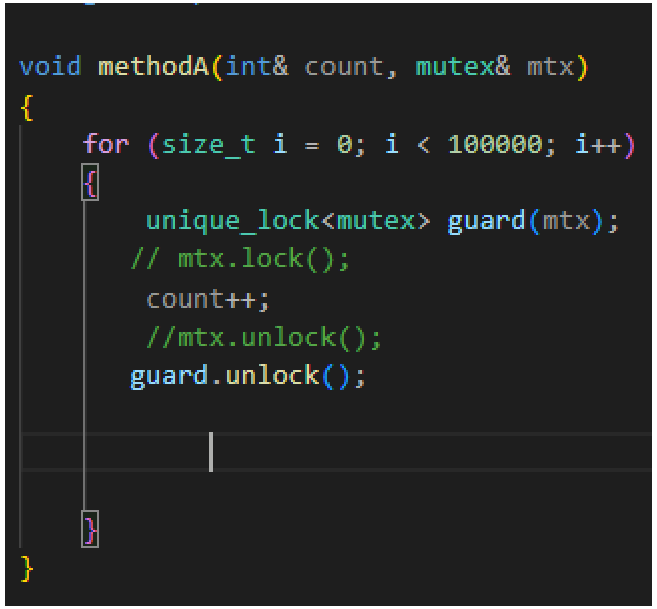
When going out of scope,
Lock guard releases the mutex it is owning.

Non-copyable

Scoped_lock : from c++17 onwards, can use with several mutexes.

https://en.cppreference.com/w/cpp/thread/lock_guard
https://en.cppreference.com/w/cpp/thread/scoped_lock

Unique_lock



```
void methodA(int& count, mutex& mtx)
{
    for (size_t i = 0; i < 100000; i++)
    {
        unique_lock<mutex> guard(mtx);
       // mtx.lock();
        count++;
        //mtx.unlock();
       guard.unlock();


    }
}
```

#include <mutex>

Light weight wrapper around mutex which make sure RAII

+ Allow

- ✓ deferred locking
- ✓ time-constrained attempts at locking
- ✓ recursive locking
- ✓ transfer of lock ownership
- ✓ use with condition variables

When going out of scope,
releases the mutex if it is owning still.

Non-copyable

Scoped_lock : from c++17 onwards, can use with several mutexes.

Deadlock

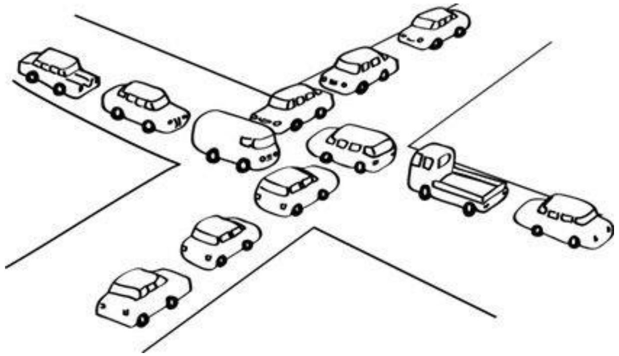
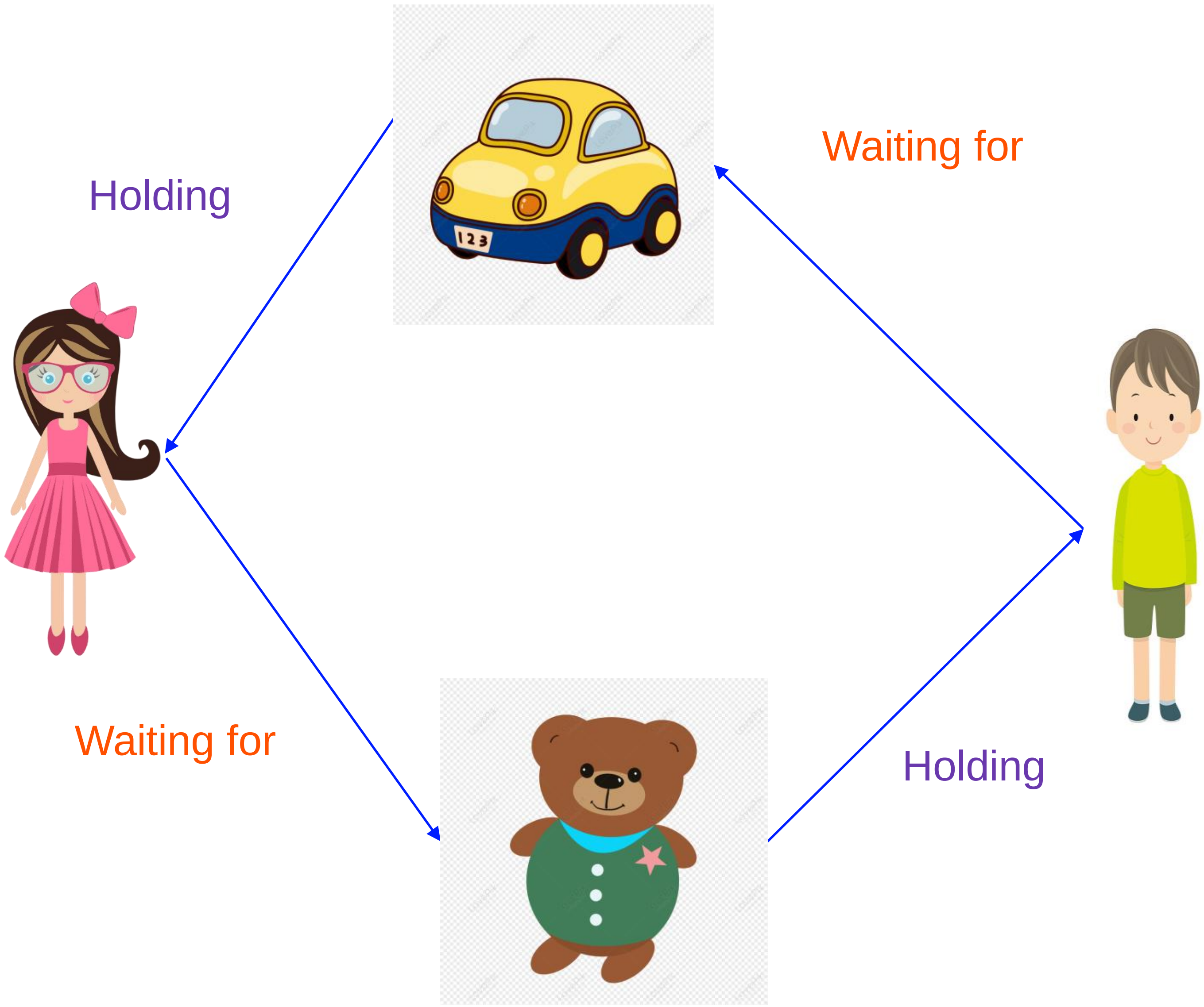


Image courtesy: https://arctype.com/blog/database-deadlock/



Waiting for

Holding

Waiting for

Holding

Deadlock

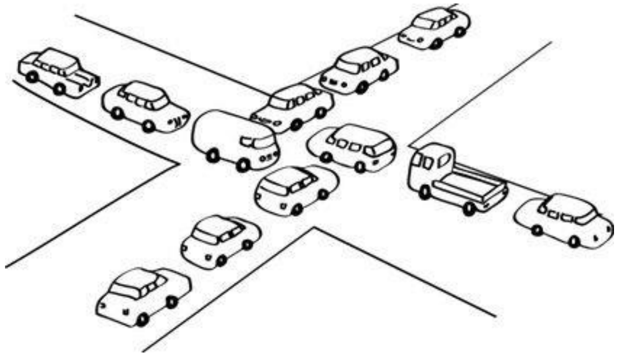
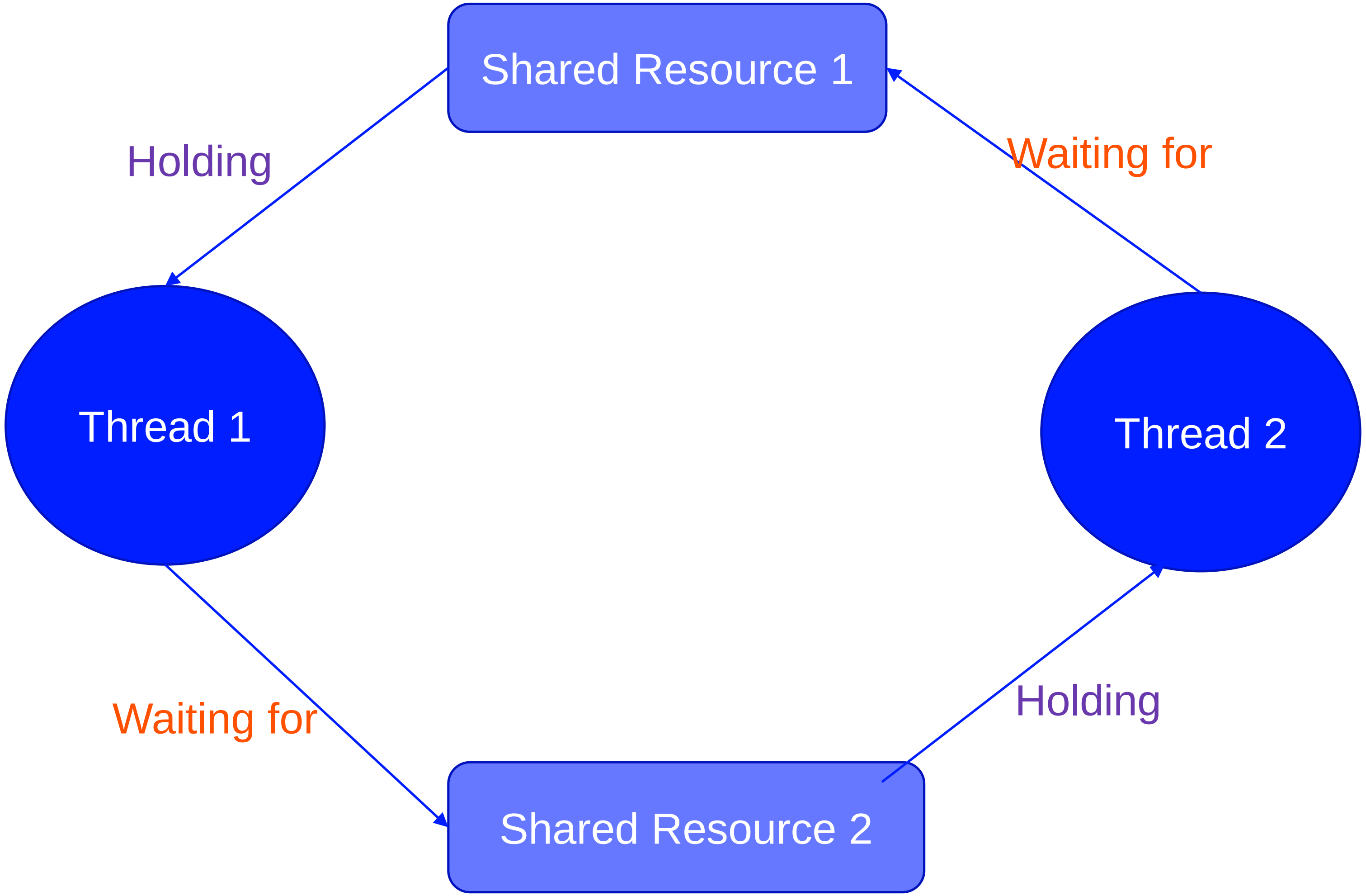
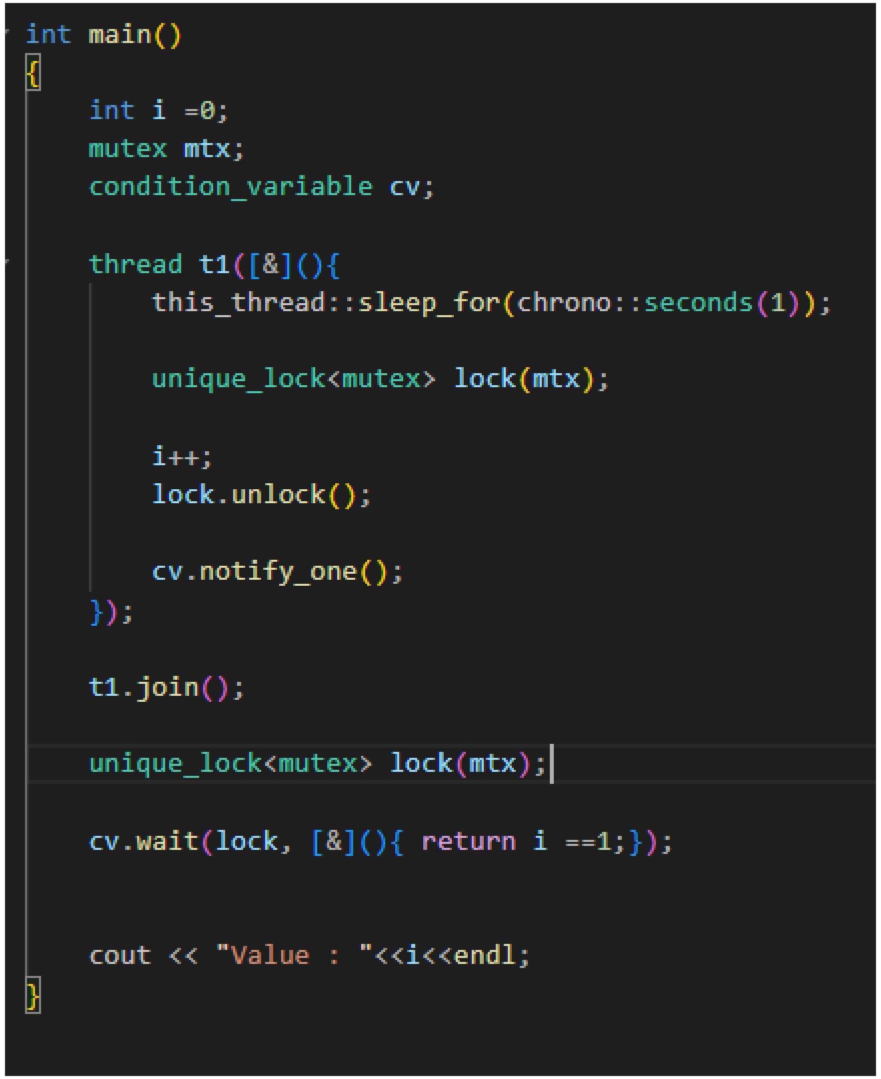


Image courtesy: https://arctype.com/blog/database-deadlock/



Condition Variable



```
int main()
{
    int i =0;
    mutex mtx;
    condition_variable cv;

    thread t1([&](){
        this_thread::sleep_for(chrono::seconds(1));

        unique_lock<mutex> lock(mtx);

        i++;
        lock.unlock();

        cv.notify_one();
    });

    t1.join();

    unique_lock<mutex> lock(mtx);

    cv.wait(lock, [&](){ return i ==1;});


    cout << "Value : "<<i<<endl;
}
```

#include <condition_variable>

Allow multiple threads to communicate with each other

Wait for 1/ more threads until 1 thread notifies

Always associated with a mutex

https://en.cppreference.com/w/cpp/thread/condition_variable

Semaphores | Latches | Barriers (C++20)

Semaphores:

#include <semaphore>

To constrain concurrent access to a shared resource

Counting_semaphore : a non-negative resource count

Binary_semaphore: only 2 states

https://en.cppreference.com/w/cpp/thread/counting_semaphore

Latches:

#include <latch>

Coorination mechanism to block until a given number of threads arrive at a given stage

Cannot resue

https://en.cppreference.com/w/cpp/thread/latch

Barriers:

#include <barrier>

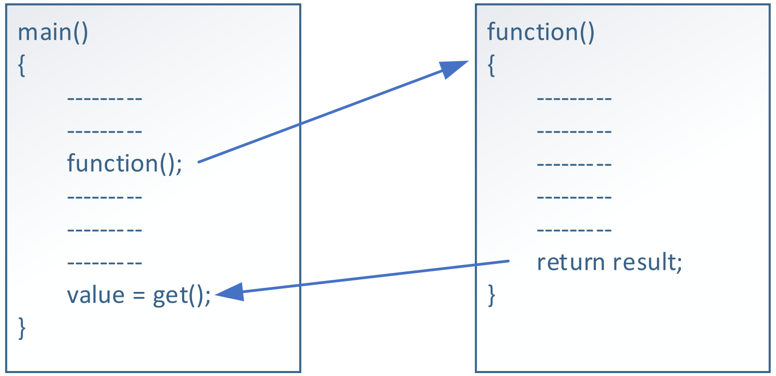
Coorination mechanism to block until a given number of threads arrive at a given stage

Can resue

https://en.cppreference.com/w/cpp/thread/barrier

std::future and std::promise

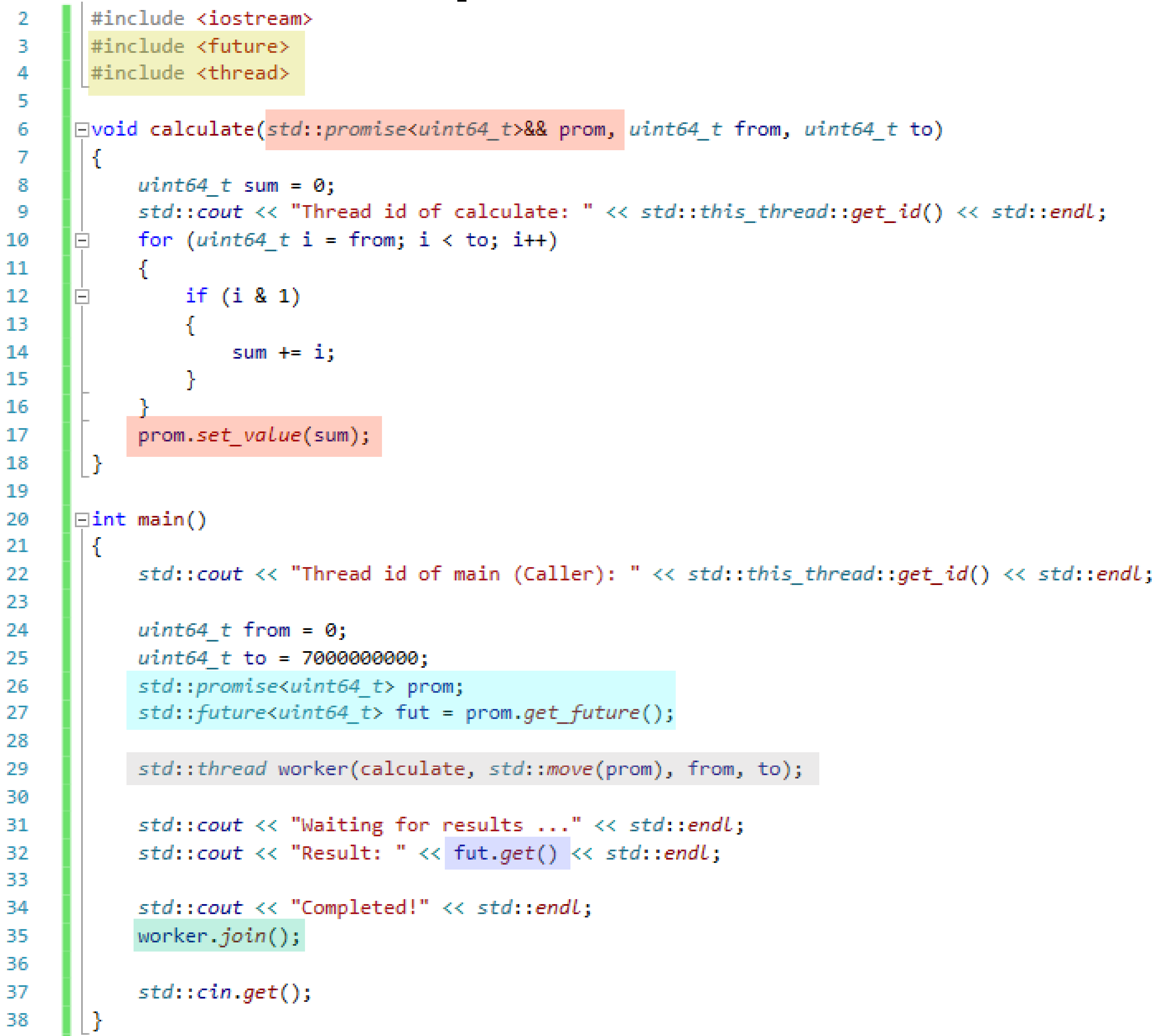
- Provides a mechanism to access the result of asynchronous operations
- std::promise
- use to set values or exceptions
- std::future
- used to get value from promise
- wait for the promise



```
main()
{
        ---------
        ---------
        function();
        ---------
        ---------
        ---------
        value = get();
}
```

```
function()
{
        ---------
        ---------
        ---------
        ---------
        ---------
        return result;
}
```

std::future and std::promise

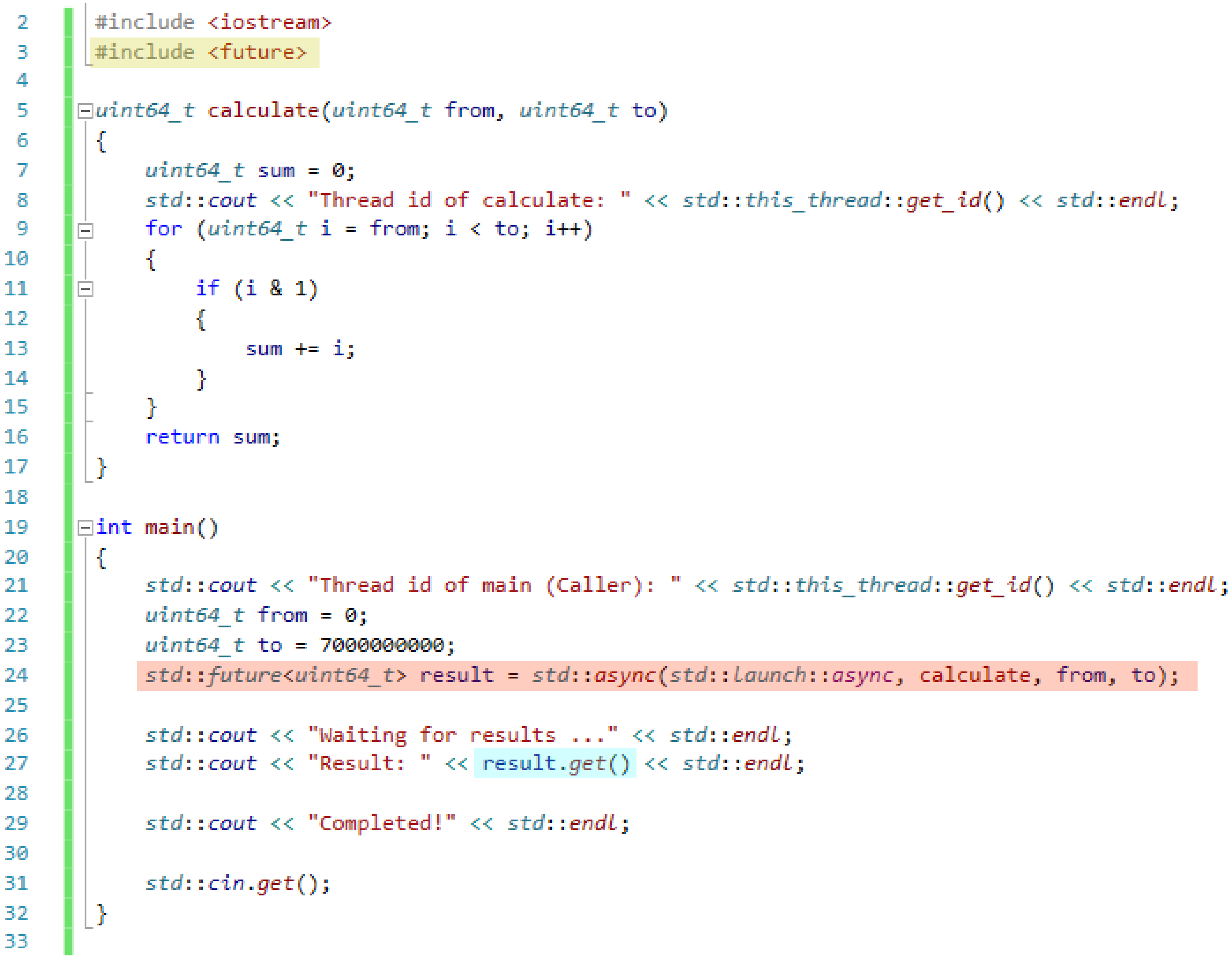


```cpp
#include <iostream>
#include <future>
#include <thread>

void calculate(std::promise<uint64_t>&& prom, uint64_t from, uint64_t to)
{
    uint64_t sum = 0;
    std::cout << "Thread id of calculate: " << std::this_thread::get_id() << std::endl;
    for (uint64_t i = from; i < to; i++)
    {
        if (i & 1)
        {
            sum += i;
        }
    }
    prom.set_value(sum);
}

int main()
{
    std::cout << "Thread id of main (Caller): " << std::this_thread::get_id() << std::endl;

    uint64_t from = 0;
    uint64_t to = 7000000000;
    std::promise<uint64_t> prom;
    std::future<uint64_t> fut = prom.get_future();

    std::thread worker(calculate, std::move(prom), from, to);

    std::cout << "Waiting for results ..." << std::endl;
    std::cout << "Result: " << fut.get() << std::endl;

    std::cout << "Completed!" << std::endl;
    worker.join();

    std::cin.get();
}
```

std::async

- Runs a function asynchronously and returns std::future
- Lunch policies
- std::launch::async
- std::launch::deferred
- std::launch::async | std::launch::deferred
- Automatically creates a thread or take from internal pool and create std::promise object
- Pass the std::promis object to thread and return std::future object

std::async



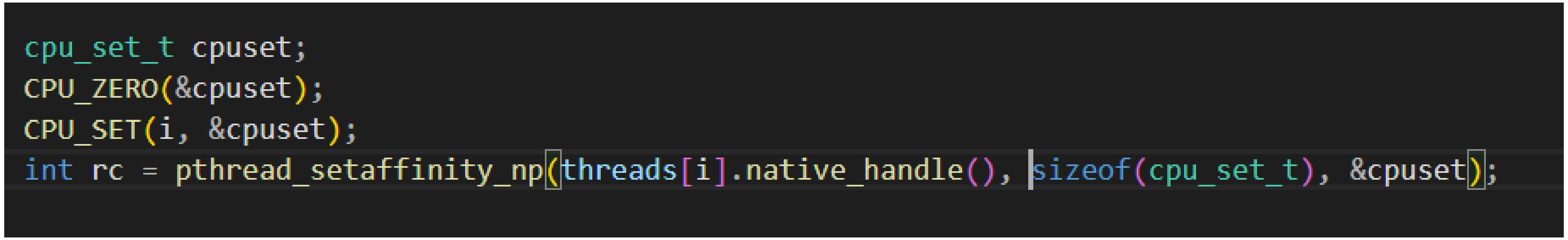
```cpp
#include <iostream>
#include <future>

uint64_t calculate(uint64_t from, uint64_t to)
{
    uint64_t sum = 0;
    std::cout << "Thread id of calculate: " << std::this_thread::get_id() << std::endl;
    for (uint64_t i = from; i < to; i++)
    {
        if (i & 1)
        {
            sum += i;
        }
    }
    return sum;
}

int main()
{
    std::cout << "Thread id of main (Caller): " << std::this_thread::get_id() << std::endl;
    uint64_t from = 0;
    uint64_t to = 7000000000;
    std::future<uint64_t> result = std::async(std::launch::async, calculate, from, to);

    std::cout << "Waiting for results ..." << std::endl;
    std::cout << "Result: " << result.get() << std::endl;

    std::cout << "Completed!" << std::endl;

    std::cin.get();
}

```

Thread Binding

thread::hardware_concurrency() : how many logical CPUs we have

Affinity: ask OS scheduler to run the given thread only in the pre-defined set of CPUs.

Pthread_setaffinity_np for linux
(https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html)



```
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(i, &cpuset);
int rc = pthread_setaffinity_np(threads[i].native_handle(), sizeof(cpu_set_t), &cpuset);
```

Interesting read: https://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/

Some Links to learn about concurrent programming

https://begriffs.com/posts/2020-03-23-concurrent-programming.html

https://www.toptal.com/software/introduction-to-concurrent-programming

https://youtu.be/LOfGJcVnvAk

THANK YOU