MULTITHREADING

ATOMIC VARIABLES

CALL ONCE

CODERS SCHOOL

BARTOSZ SZURGOT

ŁUKASZ ZIOBROŃ

AGENDA

1. Thread-safe queue
2. `std::condition_variable`
3. Memory model
4. How to stop a looped thread?
5. Atomic variables
6. `std::memory_order`
7. `std::call_once`
8. Thread-safe singletons
9. Exceptions in `std::call_once`

C++ MEMORY MODEL

C++ MEMORY MODEL

- The smallest memory unit is 1 byte
- Each byte has a unique memory address
- Synchronization is not needed if you are writing something concurrently to different areas of memory

```
vector<int> v{10};
thread t1([&]{ v[0] = 5; });
thread t2([&]{ v[1] = 15; });
```

- Synchronization is needed when you are writing something concurrently to the same memory areas
- Synchronization is needed if at least one thread is writing and others are reading the same area of memory
- No synchronization when required == race condition == undefined behavior
- `const` implies multi-threaded safety, because it guarantees reading only
- C++ Memory model on cppreference.com

IS SYNCHRONIZATION NEEDED HERE?

```
struct S {
    char a;
    int b;
} obj;
thread t1([&]{ obj.a = 'a'; });
thread t2([&]{ obj.b = 4; });
```

- Since C++11 no. Despite the same structure, the memory areas in which we save data are separate
- Race condition is possible in old POSIX threads

IS SYNCHRONIZATION NEEDED HERE?

```
vector<int> v(10, 0);
for (int = 0; i < 10; i++)
    thread t([&]{ v[i] = i; });
```

- YES
- There is a data race on variable `i`
- But the vector access is race free
- Despite the same structure, the memory areas in which we save data are separate
- Proper version that do not require synchronization:

```
vector<int> v(10, 0);
for (int = 0; i < 10; i++)
   thread t([&, i]{ v[i] = i; });
```

IS SYNCHRONIZATION NEEDED HERE?

```cpp
vector<int> v;
for (int = 0; i < 10; i++)
    thread t([&]{ v.emplace_back(i); });
```

- YES
- We have to increment the `end()` iterator while we are adding new elements - possible race condition
- When adding a new object, the vector may be reallocated. Some threads may have iterators on the deprecated vector position.

HOW TO SYNCHRONIZE?

How to sync writing / writing + reading?

- `std::mutex`

```
int a = 0;
mutex m;
thread t1([&]{
    lock_guard<mutex> lg(m);
    a = 1;
});
thread t2([&]{
    lock_guard<mutex> lg(m);
    a = 2;
});
```

- `std::atomic<T>`

```
atomic<int> a = 0;
thread t1([&]{ a = 1; });
thread t2([&]{ a = 2; });
```

LOOPED THREAD

HOW TO STOP A LOOPED THREAD?

```cpp
#include <thread>
using namespace std;

int main() {
    bool stop = false;
    auto f = [&] {
        while (not stop) {
            /* do sth... */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```


```
$> g++ 01_stop.cpp -lpthread -fsanitize=thread
$> ./a.out
WARNING: ThreadSanitizer: data race (pid=10179)
...
$> g++ 01_stop.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
deadlock
```

HOW TO STOP A LOOPED THREAD?

VOLATILE?

```cpp
#include <thread>
using namespace std;

int main() {
    volatile bool stop = false;
    auto f = [&] {
        while (not stop) {
            /* do sth... */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```


```
$> g++ 01b_volatile.cpp -lpthread -fsanitize=thread
$> ./a.out
WARNING: ThreadSanitizer: data race (pid=10179)
...
$> g++ 01b_volatile.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
```

```
WARNING: ThreadSanitizer: data race (pid=10179)
...
```

HOW TO STOP A LOOPED THREAD?

VARIABLE WITH MUTEX?

```cpp
#include <thread>
#include <mutex>

using namespace std;

int main() {
    bool flag = false;
    mutex m;

    auto stop = [&] {
        lock_guard<mutex> lg(m);
        return flag;
    };

    auto f = [&] {
        while (not stop()) {
            /* do sth... */
        }
    };
    thread t(f);
    {
        lock_guard<mutex> lg(m);
        flag = true;
    }
    t.join();
    return 0;
}
```


```
$> g++ 01c_mutex.cpp -lpthread -fsanitize=thread
$> ./a.out
$> g++ 01c_mutex.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
```

HOW TO STOP A LOOPED THREAD?

ATOMIC VARIABLE

```cpp
#include <thread>
#include <atomic>
using namespace std;

int main() {
    atomic<bool> stop{false};
    auto f = [&] {
        while (not stop) {
            /* do sth... */
        }
    };
    thread t(f);
    stop = true;
    t.join();
    return 0;
}
```

```
$> g++ 01d_atomic.cpp -lpthread -fsanitize=thread -O3
$> ./a.out
```

std::memory_order

NORMAL VARIABLES VS ATOMIC VARIABLES

Normal variables

- simultaneous write and read == undefined behavior
- need to block with mutexes in case of modification

Atomic variables

- simultaneous write and read == defined behavior
- no additional locking mechanisms required

ATOMIC VARIABLES

THE MOST IMPORTANT OPERATIONS

- `store()` - saves the value in an atomic variable, you can also specify `std::memory_order`
- `operator=()` - saves the value in an atomic variable
- `load()` - reads the value from an atomic variable, you can also specify `std::memory_order`
- `operator T()` - reads the value from an atomic variable

`std::memory_order`

- As part of optimization, the compiler has the right to change the order of operations
- The compiler needs to know which operations can be rearranged and which must follow a certain order
- SC - Sequential consistency (`memory_order_seq_cst`) ensures that the order of operations set by the developer is preserved, sometimes at the expense of performance. This is the default behavior of `std::atomic` variables
- Thanks to SC, we can correctly conclude what values may variables have regardless of the processor optimization
  - Compiler optimizations cannot create race conditions
  - Boring Details: memory_order on cppreference.com

SEQUENTIAL CONSISTENCY AND OPTIMIZATIONS

```
// INPUT:
int foo(int a)
{
    if(a<1)
        b=2;
    if(a==2)
        b=2;
    if(a>2)
        b=2;
    return b;
}
```

```
// OPT1:
int foo(int a)
{
    if(a>2)
        b=2;
    else
        if(a<1)
            b=2;
        else
            if(a==2)
                b=2;
    return b;
}
```

```
// OPT2:
int foo(int a)
{
    const int tmp=b;
    b=2;
    if(a==1)
        b=tmp;
    return b;
}
```

ARE OPT1 AND OPT2 CORRECT?

- Only OPT1
- In OPT2, state b has changed regardless of the value of a
- Another thread may have read b value at this time

`std::memory_order` VALUES

- `memory_order_relaxed`
- `memory_order_consume`
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel`
- `memory_order_seq_cst`

memory_order_relaxed

- No synchronization or ordering constraints imposed on other reads or writes
- Only this operation's atomicity is guaranteed (read or write)

```cpp
int main() {
    std::atomic<int> x{}, y{};
    int r1{}, r2{};

    std::thread t1{[&] {
        r1 = y.load(std::memory_order_relaxed); // A
        x.store(r1, std::memory_order_relaxed); // B
    }};
    std::thread t2{[&] {
        r2 = x.load(std::memory_order_relaxed); // C
        y.store(42, std::memory_order_relaxed); // D
    }};
    t1.join();
    t2.join();
    std::cout << r1 << ' ' << r2 << '\n';
}
```

POSSIBLE RESULTS?

memory_order_relaxed

```
std::thread t1{[&] {
    r1 = y.load(std::memory_order_relaxed); // A
    x.store(r1, std::memory_order_relaxed); // B
}};
std::thread t2{[&] {
    r2 = x.load(std::memory_order_relaxed); // C
    y.store(42, std::memory_order_relaxed); // D
}};
```

POSSIBLE RESULTS

- `r1 = 0, r2 = 0`
  - orders of execution: ABCD, ACBD, ACDB, CADB, CABD
- `r1 = 42, r2 = 0`
  - orders of execution: CDAB
- `r1 = 42, r2 = 42`
  - orders of execution: DABC !?

HOW IS IT POSSIBLE?

memory_order_relaxed

```
std::thread t1{[&] {
    r1 = y.load(std::memory_order_relaxed); // A
    x.store(r1, std::memory_order_relaxed); // B
}};
std::thread t2{[&] {
    r2 = x.load(std::memory_order_relaxed); // C
    y.store(42, std::memory_order_relaxed); // D
}};
```

HOW IS IT POSSIBLE?

- Compiler reordering!
- A is sequenced-before B within thread 1 and A must occur before B. B depends on the result of A.
- C is sequenced-before D within thread 2, but they are independent.
- Nothing prevents D from appearing before A in the modification order of y, and B from appearing before C in the modification order of x

SEQUENTIAL CONSISTENCY (CS)

- Aka "full barrier", "standalone fence", "full memory fence"
- Not only order memory the same way as release/acquire ordering, but also establish a single total modification order of all atomic operations that are so tagged.

SEQUENTIAL CONSISTENCY

```
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

v1 = 5;
v2 = 10;
auto value = number.load(std::memory_order_seq_cst); // full barrier
v2 = 15;
v1 = 20;
number.store(10, std::memory_order_seq_cst);         // full barrier
v2 = 25;
v3 = 30;
```

- Instructions can't be reordered across full barriers

SEQUENTIAL CONSISTENCY

```cpp
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

v2 = 10;
v1 = 5;
auto value = number.load(std::memory_order_seq_cst); // full barrier
v1 = 20;
v2 = 15;
number.store(10, std::memory_order_seq_cst);         // full barrier
v3 = 30;
v2 = 25;
```

- Instructions can't be reordered across full barriers

SEQUENTIAL CONSISTENCY

```cpp
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

v2 = 10;
v1 = 5;
auto value = number.load(); // full barrier
v1 = 20;
v2 = 15;
number.store(10);           // full barrier
v3 = 30;
v2 = 25;
```

- Instructions can't be reordered across full barriers
- `std::memory_order_seq_cst` is a default value for `load()` and `store()`

SEQUENTIAL CONSISTENCY

```
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

v2 = 10;
v1 = 5;
int value = number; // full barrier
v1 = 20;
v2 = 15;
number = 10;        // full barrier
v3 = 30;
v2 = 25;
```

- Instructions can't be reordered across full barriers
- `std::memory_order_seq_cst` is a default value for `load()` and `store()`
- You can also read data with conversion `operator T()`
- And write data with assignment `operator=()`

SEQUENTIAL CONSISTENCY

- Sequential ordering may be necessary for multiple producer-multiple consumer situations where all consumers must observe the actions of all producers occurring in the same order.
- SC may become a performance bottleneck since it forces the affected memory accesses to propagate to every CPU core.
- Conceptually, there is single global memory and a "switch" that connects an arbitrary processor to memory at any time step

`memory_order_acquire` / `memory_order_release`

- Aka "one-way barriers", "acquire/release fences", "half fences"
- A release store makes its prior access visible to thread performing an acquire load that sees that store.

ACQUIRE/RELEASE

```cpp
int v1 = 0;
int v2 = 0;
std::mutex m;
bool b = false;

v1 = 5;
m.lock();    // acquire ⬇️✅  ⬆️❌
b = true;
m.unlock(); // release ⬆️✅  ⬇️❌
v2 = 10;
```

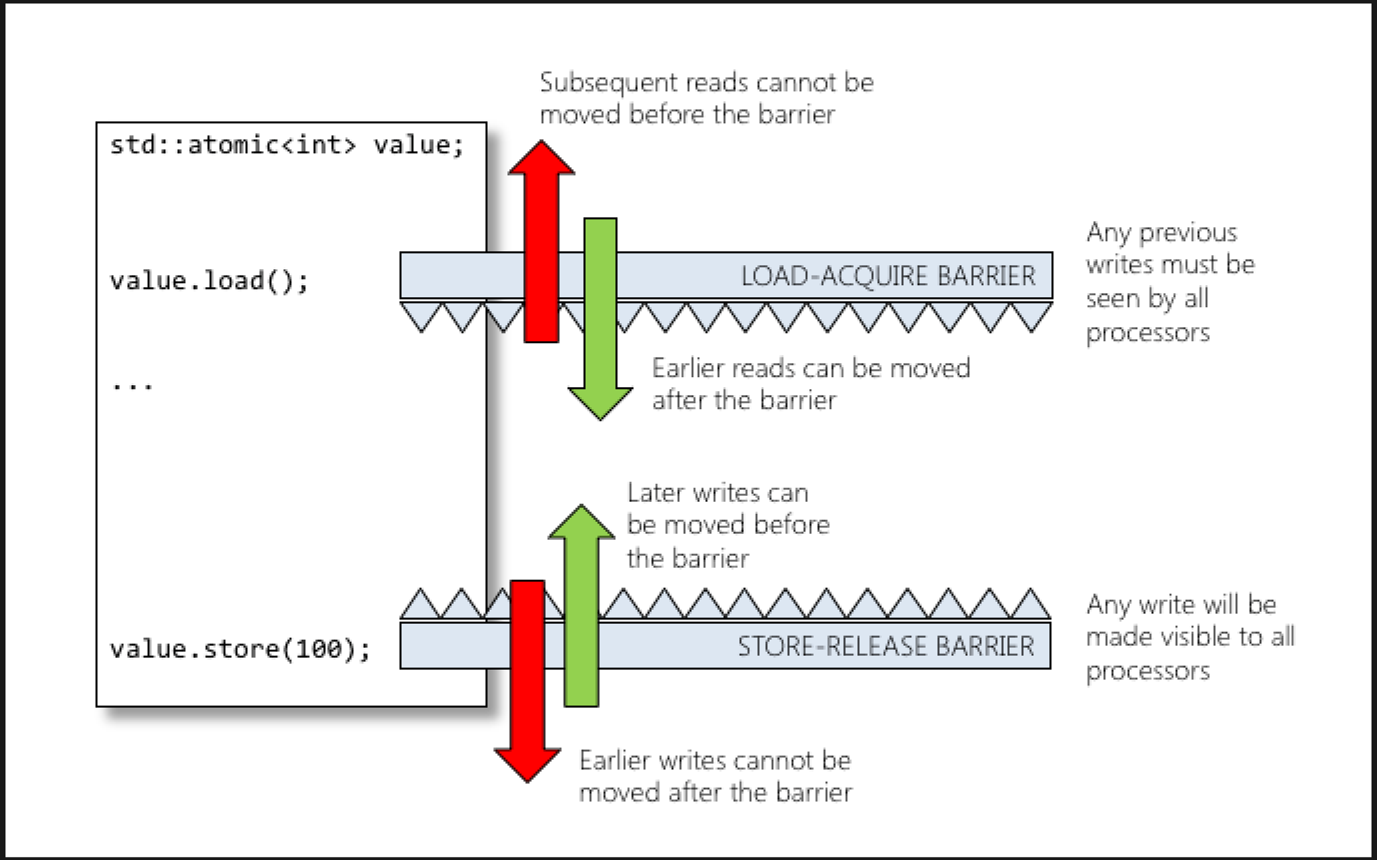
ACQUIRE/RELEASE

```cpp
int v1 = 0;
int v2 = 0;
std::mutex m;
bool b = false;

m.lock();    // acquire ⬇️✅  ⬆️❌
v1 = 5;
b = true;
v2 = 10;
m.unlock(); // release ⬆️✅  ⬇️❌
```

You can always move code inside a global section, but not outside

ACQUIRE/RELEASE



Source

ACQUIRE/RELEASE

```cpp
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

v1 = 5;
v2 = 10;
auto value = number.load(std::memory_order_acquire); // ⬇️⬇️⬇️
v2 = 15;
v1 = 20;
number.store(10, std::memory_order_release);         // ⬆️⬆️⬆️
v2 = 25;
v3 = 30;
```

- Instructions can be reordered only in one way
- Acquire pass instruction from above
- Release pass instruction from below

ACQUIRE/RELEASE

```cpp
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

auto value = number.load(std::memory_order_acquire); // ⬇️⬇️⬇️
v1 = 5;
v2 = 10;
v2 = 15;
v1 = 20;
v2 = 25;
v3 = 30;
number.store(10, std::memory_order_release);         // ⬆️⬆️⬆️
```

- Instructions can be reordered only in one way
- Acquire pass instruction from above
- Release pass instruction from below

ACQUIRE/RELEASE

```cpp
int v1 = 0;
int v2 = 0;
int v3 = 0;
std::atomic<int> number{0};

auto value = number.load(std::memory_order_acquire); // ⬇️⬇️⬇️
v1 = 20;
v2 = 25;
v3 = 30;
number.store(10, std::memory_order_release);         // ⬆️⬆️⬆️
```

- Instructions can be reordered only in one way
- Acquire pass instruction from above
- Release pass instruction from below
- Some compiler optimizations are possible :)

ACQUIRE/RELEASE

- Acquire: no reads or writes in the current thread can be reordered before this load.
  - Acquire allows earlier instruction to be executed later
  - `atomic.load()` -> use acquire (the default is seq_cst)
- Release: no reads or writes in the current thread can be reordered after this store.
  - Release allows later instructions to be executed earlier
  - `atomic.store()` -> use release (the default is seq_cst)
- `std::mutex` is an example of release-acquire synchronization: when the lock is released by thread A and acquired by thread B, everything that took place in the critical section (before the release) in the context of thread A has to be visible to thread B (after the acquire).
  - `mutex.lock()` == acquire
  - `mutex.unlock()` == release

EXAMPLES

- `memory_order_relaxed` ↕️↕️↕️
  - In general - counters incrementation
  - `shared_ptr`'s reference counter incrementation (but not decrementation!)
- `memory_order_acquire` ⬇️⬇️⬇️
  - `mutex.lock()`
  - `atomic.load()` (the default is seq_cst, but you can safely use acquire)
- `memory_order_release` ⬆️⬆️⬆️
  - `mutex.unlock()`
  - `atomic.store()` (the default is seq_cst, but you can safely use release)
- `memory_order_seq_cst` ❌❌❌
  - multiple producer-multiple consumer situations

std::memory_order

There are 2 more:

- memory_order_consume 🐉🐉🐉
  - "Dragons be here"
  - std::kill_dependency and [[carries_dependency]]
  - This is too difficult...
  - Must be 3m (10 feet) tall to use it 😉
- memory_order_acq_rel
  - Both acquire and release fences in one sequence point = full barrier
  - But it's not sequentially consistent
  - Used in instructions that read and write data atomically - compare_exchange_strong, compare_exchange_weak, exchange
  - shared_ptr's reference counter decrementation (watch Atomic Weapons by Herb Sutter)

ATOMIC VARIABLES - SUMMARY

- `std::atomic` is a light synchronization
- Allows simple arithmetic and bitwise operations: ++, --, +=, -=, &=, |=, ^=
- Typically: numbers, pointers
- Uses special processor instructions to modify data atomically
- `std::atomic` doesn't make sense on complex types
  - there are no special processor instructions that ensure the indivisibility of such operations
  - no transactional memory model in C++ (yet)
  - if it is successful, it may not work as intended (see Stack Overflow)
  - you must use mutexes
- The default memory order is sequential consistency. Avoid it.
  - Prefer acquire for reading data
  - Prefer release for writing data
  - Rather do not use relaxed memory order. It may be useful only in rare situations (reference counting)
  - But remember, that clear code is better than prematurely optimized code 😉

Atomic<> Weapons by Herb Sutter

What do each memory_order mean? - StackOverflow

std::call_once

`std::call_once`

- `#include <mutex>`
- `std::call_once`
- Wraps a function that must be executed only once
- Guarantees one call, even when called concurrently by several threads
- Calls the passed function on its own thread (doesn't create a new one)
- It needs `std::once_flag` flag

std::call_once

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

std::once_flag flag;

void do_once() {
    std::call_once(flag, [] {
        std::cout << "Called once!\n";
    });
}

int main() {
    std::thread t1(do_once);
    std::thread t2(do_once);
    std::thread t3(do_once);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

```
$> g++ 01_call_once.cpp -lpthread -fsanitize=thread
$> ./a.out
Called once!
```

ONCE_FLAG

- `#include <mutex>`
- `std::once_flag`
- An auxiliary structure for use with `std::call_once`
- Neither copyable nor movable
- Contains information whether the function has already been called
- The constructor sets the state to not called

`call_once` WAY OF OPERATION

- If `once_flag` is in the "called" state, `call_once` immediately returns (passive call)
- If `once_flag` is in the "not called" state, `call_once` executes the passed function, passing further arguments to it (active call)
  - If the function throws an exception, it is propagated on, and `once_flag` is not set to "called state" (exceptional call), so different `call_once` can be called (at least in theory 🙂)
  - If the function ends normally, `once_flag` is set to the "called" state. It is guaranteed that all other calls will be passive.
- Multiple active calls on the same `once_flag` are queued.
- If the same flag is used for concurrent calls to different functions, it is not specified which function will be called.

EXERCISE: RACE

```cpp
void setWinner() {
    auto id = this_thread::get_id();
    auto sleepDuration = dist(rng);
    stringstream msg;
    msg << "Called " << __FUNCTION__
        << "(" << id << "). Chasing time: "
        << sleepDuration << "ms\n";
    cout << msg.str();
    this_thread::sleep_for(chrono::milliseconds(sleepDura

    // TODO: set me as a winner but don't let others ove
}
```

`exercises/05_race.cpp`

- 10 contestants (threads) are racing for $1 million
- Only the first player wins the prize, the rest will get nothing
- Implement the function `setWinner()` so that the winning thread sets itself as the winner and does not allow others to override this value.

```
$> g++ 01_race.cpp -lpthread -fsanitize=thread
$> ./a.out
Called setWinner(139887531521792). Sleeping for 15ms
Called setWinner(139887523129088). Sleeping for 35ms
Called setWinner(139887497950976). Sleeping for 31ms
Call once for 139887531521792
Called setWinner(139887489558272). Sleeping for 16ms
Called setWinner(139887481165568). Sleeping for 14ms
Called setWinner(139887453927168). Sleeping for 35ms
And the winner is... 139887531521792
```

EXERCISE - SOLUTION

```cpp
void setWinner() {
    auto id = this_thread::get_id();
    auto sleepDuration = dist(rng);
    stringstream msg;
    msg << "Called " << __FUNCTION__
        << "(" << id << "). Chasing time: "
        << sleepDuration << "ms\n";
    cout << msg.str();
    this_thread::sleep_for(chrono::milliseconds(sleepDuration));

    call_once(once, [&]{
        cout << "Call once for " << id << '\n';
        std::stringstream troublesomeConversion;
        troublesomeConversion << id;
        winnerId = troublesomeConversion.str();
    });
}
```

EXERCISE: MUTUALLY EXCLUSIVE CALLS

exercises/06_exclusive_calls.cpp

```
class X {
    vector<double> values;

    void initializeOne() { values = {1.0}; }
    void initializeTwo() { values = {1.0, 2.0}; }
    void initializeThree() { values = {1.0, 2.0, 3.(

public:
    explicit X(int i) noexcept {
        switch (i) {
        case 2: // top priority
            initializeTwo();
            [[fallthrough]];
        case 3:
            initializeThree();
            [[fallthrough]];
        default: // least priority
            initializeOne();
        }
    }
    // ...
};
```

- Add the appropriate `call_once` and messages so that the output appears as below
- Do not modify the constructor 😉

```
$> g++ 02_exclusive_calls.cpp
-lpthread -fsanitize=thread
$> ./a.out
initializeTwo
Call once initializeTwo
initializeThree
initializeOne
1 2
initializeThree
Call once initializeThree
initializeOne
1 2 3
initializeOne
Call once initializeOne
1
```

EXERCISE - SOLUTION

```
class X {
    once_flag once;
    vector<double> values;

    void initializeOne() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeOne\n";
            values = {1.0};
        });
    }

    void initializeTwo() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeTwo\n";
            values = {1.0, 2.0};
        });
    }

    void initializeThree() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
```

THREAD-SAFE SINGLETONS

CODERS SCHOOL

EXAMPLE: THREAD-SAFE SINGLETON

```cpp
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    static std::mutex mutex_;
    Singleton() = default;
public:
    static Singleton& getInstance() {
        {
            std::unique_lock<std::mutex> lock(mutex_);
            if (!instance_) {
                instance_.reset(new Singleton{});
            }
        }
        return *instance_;
    }
};
```

- Slow (mutexes)
- Safe
- Long Code

EXAMPLE: THREAD-SAFE SINGLETON

```
class Singleton {
    static std::unique_ptr<Singleton> instance_;
    static std::once_flag flag_;
    Singleton() = default;
public:
    static Singleton& getInstance() {
        std::call_once(flag_, [&] {
            instance_.reset(new Singleton{});
        });
        return *instance_;
    }
};
```

- Slow (once_flag)
- Safe
- Less code

EXAMPLE: THREAD-SAFE SINGLETON

```cpp
class Singleton {
    Singleton() = default;

public:
    static Singleton& getInstance() {
        static Singleton instance_;
        return instance_;
    }
};

// Meyers Singleton
```

- The fastest one
- Safe - static initialization is thread-safe since C++11
- Short
- Lovely

EXCEPTIONS IN CALL_ONCE

EXERCISE: EXCEPTIONS IN CALL_ONCE

exercises/07_exceptional_exclusive_calls.cpp

```cpp
class X {
    once_flag once;
    vector<double> values;

    void initializeOne() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeOne\n";
            values = {1.0};
        });
    }

    // ...

    void initializePierdyliard() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializePierdyliard\n";
            throw std::bad_alloc{};
            // TODO: Can you fix me?
        });
    }
};
```

- Try to fix the problem with throwing exceptions in call_once
- According to cppreference.com:
  - If that invocation throws an exception, it is propagated to the caller of call_once, and the flag is not flipped so that another call will be attempted (such call to call_once is known as exceptional)

```
$> g++ 03_exceptional_exclusive_calls.cpp
-lpthread -fsanitize=thread
$> ./a.out
...
initializePierdyliard
Call once initializePierdyliard
terminate called after throwing an instance
of 'std::bad_alloc'
what(): std::bad_alloc
Aborted (core dumped)
```

SOLUTION

```
class X {
    once_flag once;
    vector<double> values;

    void initializeOne() {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializeOne\n";
            values = {1.0};
        });
    }

    // ...

    void initializePierdyliard() try {
        cout << __FUNCTION__ << '\n';
        call_once(once, [&]{
            cout << "Call once initializePierdyliard\n";
            throw std::bad_alloc{};
        });
    } catch (...) { /* ignore exceptions */ }
};
```

- Try to fix the problem with throwing exceptions in call_once
- According to cppreference.com:
  - If that invocation throws an exception, it is propagated to the caller of call_once, and the flag is not flipped so that another call will be attempted (such call to call_once is known as exceptional)

NOT POSSIBLE?

(AT LEAST FOR G++7.4.0 🙂)

```
$> g++ 03_exceptional_exclusive_calls.cpp
-lpthread -fsanitize=thread
$> ./a.out
...
initializePierdyliard
Call once initializePierdyliard
initializeOne
(hang up)
```

EXCEPTIONS IN CALL_ONCE - BUG IN STANDARD LIBRARY IMPLEMENTATION

- If once_flag is in the "called" state, call_once immediately returns - return (passive call)
- If once_flag is in the "not called" state, call_once executes the passed function, passing further arguments to it (active call)
  - If the function throws an exception, it is propagated on, and `once_flag` is not set to "called state" (exceptional call), so different `call_once` can be called (at least in theory 🙂) - implementation bug, example at cppreference.com also doesn't work. Supposedly works in MSVC (Visual Studio Compiler) and new versions of Apple Clang
  - If the function ends normally, once_flag is set to the "called" state. It is guaranteed that all other calls will be passive.
- Multiple active calls on the same once_flag are queued.
- If the same flag is used for concurrent calls to different functions, it is not specified which function will be called.

RECAP

1. WHAT WAS THE MOST SURPRISING FOR YOU? 😲
2. WHAT WAS THE MOST OBVIOUS FOR YOU? 🥱

POINTS TO REMEMBER

- `condition_variable`
  - Always use `wait()` with a predicate!
  - Remember about spurious wake-ups and lost notifications
  - Remember about a fight for locking a mutex when you use `notify_all()`
  - You can't choose which thread should wake up when you use `notify_one()`
- `atomic`
  - Only a single operation on atomic type is atomic. Read + write may not be atomic, if you use it incorrectly
  - You need to use proper operators to have atomic read + write, or use `fetch_*()` functions, `exchange()` or `compare_exchange_*()`
  - The default memory order for `atomic` is sequential consistency. If you need some optimizations you have to loosen the memory order by specifying it manually.
  - Memory order acquire is used with `load()` and release is used with `store`. If you exchange them, you have undefined behavior.
- `call_once`
  - Mind the ~~gcc~~ bug in g++ and clang++ - exceptional call may hang the program
  - If you use the same `once_flag` for different functions, you don't know which function will be called.

PRE-TEST 🤯

```cpp
// assume all necessary includes are here

int main() {
    std::mutex m;
    std::condition_variable cv;
    std::vector<int> v;
    std::vector<std::thread> producers;
    std::vector<std::thread> consumers;

    auto consume = [&] {
        std::unique_lock<std::mutex> ul(m);
        cv.wait(ul);
        std::cout << v.back();
        v.pop_back();
    };
    for (int i = 0; i < 10; i++) consumers.emplace_back(consume);

    auto produce = [&](int i) {
        {
            std::lock_guard<std::mutex> lg(m);
            v.push_back(i);
        }
        cv.notify_all();
    };
    for (int i = 0; i < 10; i++) producers.emplace_back(produce, i);

    for (auto && p : producers) p.join();
    for (auto && c : consumers) c.join();
}
```

1. there may be an Undefined Behavior in this code
2. the output is guaranteed to always be `0123456789`
3. `v` is always an empty vector at the end of this program
4. if some producers threads started before some consumers, we would have a deadlock because of lost notifications
5. a change from `notify_all()` to `notify_one()` guarantees that each consumer thread will receive a different number
6. this code can be improved by providing a predicate to `wait()` to disallow getting elements when the vector is empty

POST-WORK

- Ping-pong
  - difficult version - exercises/03a_ping_pong.cpp
  - easier version - exercises/03b_ping_pong.cpp
- Post-test
- Training evaluation

HOMEWORK: PING-PONG

- Thread A prints "ping" and the consecutive number
- Thread B prints "pong" and the consecutive number
- Ping always starts. Pong always ends.
- Threads must work in turns. There may not be 2 consecutive pings or pongs. The program cannot end with a ping without a respective pong.
- The program must be terminated either after the given number of repetitions or after the time limit, whichever occurs first. The reason for termination should be displayed on the screen.
- Program parameters:
  - number of repetitions
  - time limit (in seconds)

```
$> g++ 03_ping_pong.cpp -lpthread
-std=c++17 -fsanitize=thread
$> ./a.out 1 10
ping 0
pong 0
Ping reached repetitions limit
Pong reached repetitions limit
$> ./a.out 12 1
ping 0
pong 0
ping 1
pong 1
ping 2
pong 2
Timeout
```

TIPS

If you got stuck:

- You need a mutex and a condition variable in your `PingPong` class
- Wait for a condition variable with `wait_for()` in `stop()` function
- Check the number of repetitions in ping and pong threads
- Use an additional `std::atomic` variable which will tell all threads to end, when the required conditions are met.
- Ping and pong threads should be using `wait()` to check if it's their turn to work. Use an additional variable that will be used in the predicate passed to `wait()`.
- The pong thread should end the program after reaching the repetition limit

USEFUL LINKS

- C++ Atomic Types and Operations (C++ Standard)
- C++ Memory model on cppreference.com
- std::memory_order on cppreference.com
- std::call_once on cppreference.com
- std::once_flag on cppreference.com
  - STL bug in exception handling in call_once
  - call_once vs mutex on stackoverflow
- Meyers Singleton on stackoverflow
- Atomic<> Weapons by Herb Sutter

CODERS SCHOOL