Acquire Release: The Typical Misunderstanding

This lesson highlights a typical misunderstanding while using acquire-release in C++.

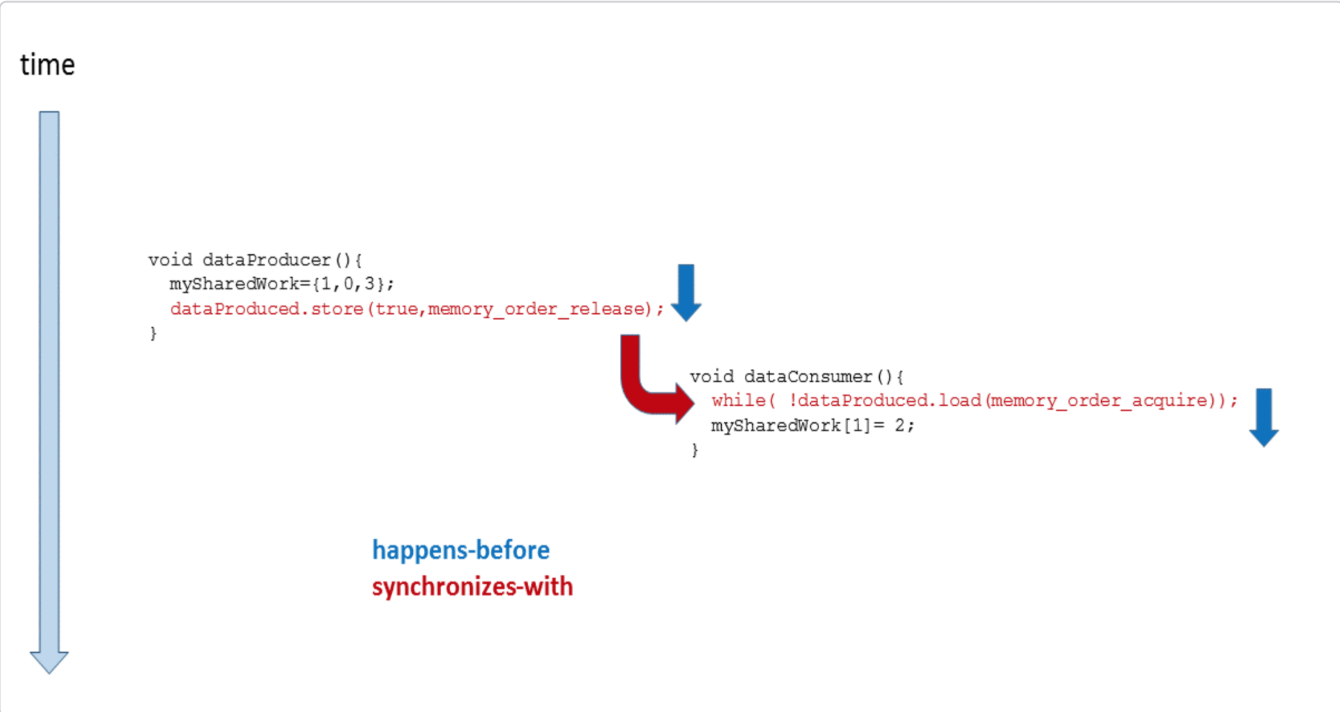
What is my motivation for writing about the typical misunderstanding of the acquire-release semantic? Many of my readers and students have already fallen into this trap. Let's look at the straightforward case. Here is a simple program as a starting point.

```
// acquireReleaseWithWaiting.cpp

#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::atomic<bool> dataProduced(false);

void dataProducer(){
    mySharedWork = {1, 0, 3};
    dataProduced.store(true, std::memory_order_release);
}

void dataConsumer(){
    while( !dataProduced.load(std::memory_order_acquire) );
    mySharedWork[1] = 2;
}

int main(){

  std::cout << std::endl;

  std::thread t1(dataConsumer);
  std::thread t2(dataProducer);

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

  for (auto v: mySharedWork){
      std::cout << v << " ";
```

```
  }

  std::cout << "\n\n";

}
```

The consumer thread `t1` in line 17 waits until the consumer thread `t2` in line 13 sets `dataProduced` to `true`. `dataProduced` is the guard and it guarantees that access to the non-atomic variable `mySharedWork` is synchronized. This means that the producer thread `t2` initializes `mySharedWork`, then the consumer thread `t1` finishes the work by setting `mySharedWork[1]` to `2`. Therefore, the program is well-defined.

The graph below shows the *happens-before* relation within the threads and the `synchronizes-with` relation between the threads; *synchronizes-with* establishes a *happens-before* relation. The rest of the reasoning is the transitivity of the *happens-before* relation: `mySharedWork = {1, 0, 3}` *happens-before* `mySharedWork[1] = 2`.



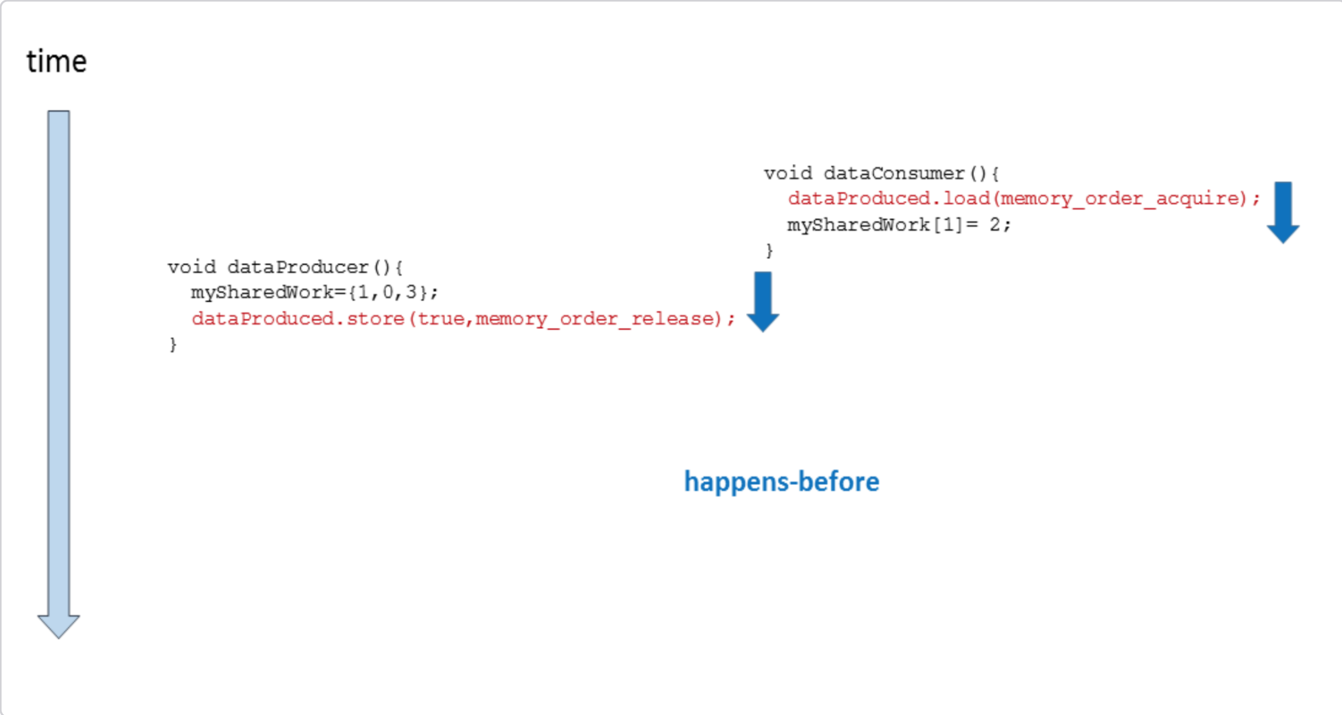
What aspect is often missing in this reasoning? The **if**. What happens if the consumer thread `t1` in line 17 doesn't wait for the producer thread?

```
// acquireReleaseWithoutWaiting.cpp

#include <atomic>
```

```
#include <iostream>
#include <thread>
#include <vector>


std::vector<int> mySharedWork;
std::atomic<bool> dataProduced(false);

void dataProducer(){
    mySharedWork = {1, 0, 3};
    dataProduced.store(true, std::memory_order_release);
}

void dataConsumer(){
    dataProduced.load(std::memory_order_acquire);
    mySharedWork[1] = 2;
}

int main(){

  std::cout << std::endl;

  std::thread t1(dataConsumer);
  std::thread t2(dataProducer);

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

  for (auto v: mySharedWork){
      std::cout << v << " ";
  }

  std::cout << "\n\n";

}
```

The program has undefined behavior because there is a data race on the variable `mySharedWork`. When we let the program run, we will get the following non-deterministic behavior.

What's the issue? It holds that `dataProduced.store(true, std::memory_order_release)` *synchronizes-with* `dataProduced.load(std::memory_order_acquire)`. However, that doesn't mean the acquire operation waits for the release operation, and that is exactly what is displayed in the graphic. In the graphic, the `dataProduced.load(std::memory_order_acquire)` instruction is performed before the instruction `dataProduced.store(true, std::memory_order_release)`. We have no *synchronizes-with* relation.



The Solution

synchronizes-with: If `dataProduced.store(true, std::memory_order_release)` happens before `dataProduced.load(std::memory_order_acquire)`, then all visible effects of the operations before `dataProduced.store(true, std::memory_order_release)` are visible after `dataProduced.load( std::memory_order_acquire)`. The key is the word **if**. That **if** will be guaranteed in the first program with the predicate `(while(!dataProduced.load(std::memory_order_acquire))`.

Now once again, but more formally:

All operations before `dataProduced.store(true, std::memory_order_release)` *happens-before* all operations after `dataProduced.load(std::memory_order_acquire)`, if the following holds : `dataProduced.store(true, std::memory_order_release)` *happens-before* `dataProduced.load(std::memory_order_acquire)`.

If you carefully follow my explanation like in the subsection Challenges, you probably expect Relaxed Semantic to come next. However, in the next lesson, I'll look first at the memory model `std::memory_order_consume` - which is quite similar to `std::memory_order_acquire`.