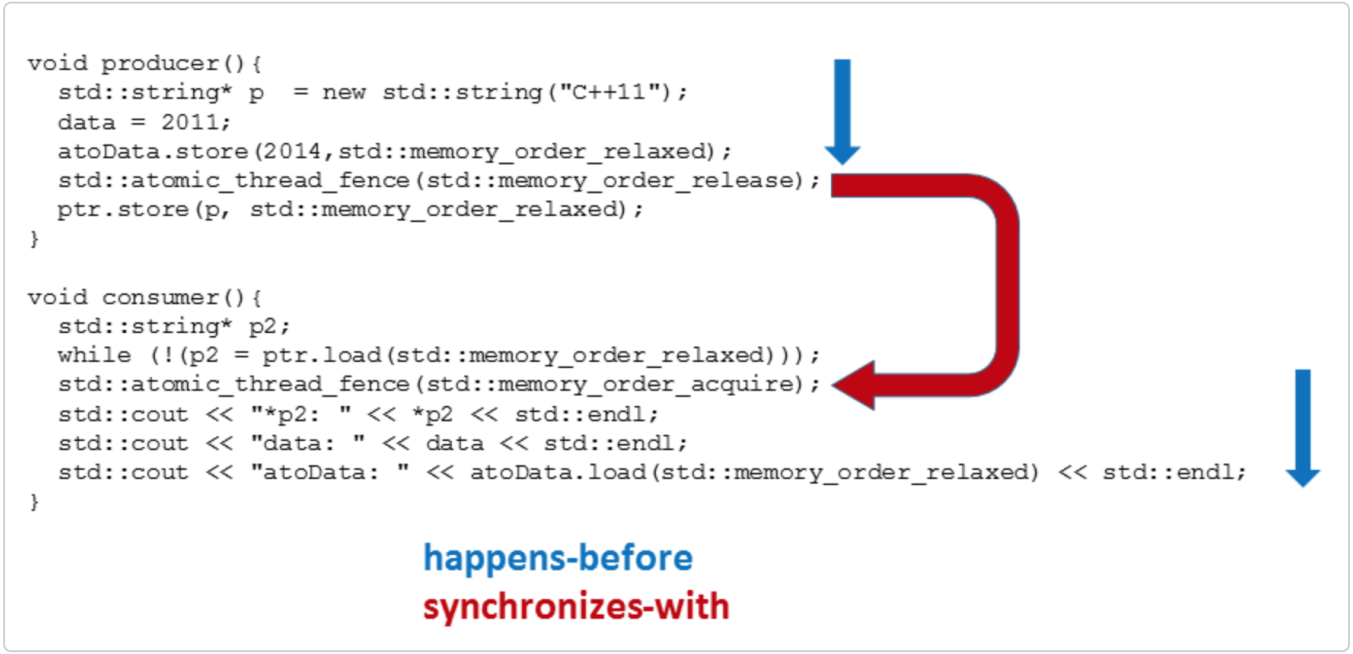
Synchronization with Fences

This lesson gives an overview of synchronization with fences in C++.

It's quite straightforward to port the program to use fences.

```
// acquireReleaseFences.cpp

#include <atomic>
#include <thread>
#include <iostream>
#include <string>

using namespace std;

atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(2014, memory_order_relaxed);
    atomic_thread_fence(memory_order_release);
    ptr.store(p, memory_order_relaxed);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_relaxed)));
    atomic_thread_fence(memory_order_acquire);
    cout << "*p2: " << *p2 << endl;
    cout << "data: " << data << endl;
    cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
}

int main(){

    cout << endl;

    thread t1(producer);
    thread t2(consumer);

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

    delete ptr;

    cout << endl;
```

The first step was to add fences with the acquire and release semantic (lines 18 and 25). Next, I changed the atomic operations with *acquire* or *release* semantic to *relaxed* semantic (lines 17 and 24) - which was straightforward. Of course, I can only replace an acquire or release operation with the corresponding fence. The key point is that the release operation with the acquire operation establishes a *synchronizes-with* relation and, therefore, a *happens-before* relation. For a more visual reader, here's the entire relation graphically.



This is the key question: Why do the operations after the acquire fence see the effects of the operations before the release fence? This is interesting because `data` is a non-atomic variable and `atoData` is used with relaxed semantic, which would suggest they can be reordered. However, thanks to the `std::atomic_thread_fence(std::memory_order_release)` as a release operation in combination with the `std::atomic_thread_fence(std::memory_order_acquire)`, neither can be reordered.

For clarity, here's the whole reasoning in a more concise form:

1. The acquire and release fences prevent the reordering of the atomic and non-atomic operations across the fences.

2. The consumer thread `t2` is waiting in the `while (!(p2=ptr.load(std::memory_order_relaxed)))` loop, until the pointer `ptr.store(p,std::memory_order_relaxed)` is set in the producer thread `t1`.
3. The release fence *synchronizes-with* the acquire fence.