

Synchronization with Atomic Variables

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

WE'LL COVER THE FOLLOWING ^

- Atomic Operations

As a starting point, I've implemented a typical consumer-producer workflow with the acquire-release semantic. Initially, I will use atomics and then will switch to fences. Let's start with atomics because most of us are comfortable with them. That will not hold for fences; they are almost completely ignored in the literature on the C++ memory model.

Atomic Operations

```
// acquireRelease.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);
    ptr.store(p, memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(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();

    cout << endl;
}

```



This program should be quite familiar to you; it is the classic example that I used in the subsection about `std::memory_order_consume`. The graphic emphasizes exactly that the consumer thread `t2` sees all values from the producer thread `t1`.

```

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

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

```

happens-before
synchronizes-with

The program is well-defined because the happens-before relation is transitive. I only have to combine the three happens-before relations:

1. Lines 14 - 16 *happens-before* line 17

`ptr.store(p, std::memory_order_release)`.

2. Line 23 `while(!(p2= ptr1.load(std::memory_order_acquire)))` *happens-before* the lines 24 - 26.

3. Line 17 *synchronizes-with* line 23. => Line 17 *happens-before* line 23.

But now the story becomes more interesting. How can I adjust the workflow to fences? We'll discuss this in the next lesson.