std::atomic<bool>

This lesson gives an overview of std::atomic<bool> which is used from the perspective of concurrency in C++.

Let's start with the full specializations for bool: `std::atomic<bool>`

`std::atomic<bool>`

`std::atomic<bool>` has a lot more to offer than `std::atomic_flag`. It can explicitly be set to `true` or `false`.

⚠ **`atomic` is not `volatile`**

What does the keyword `volatile` in C# and Java have in common with the keyword `volatile` in C++? Nothing! It's so easy in C++. That is the difference between `volatile` and `std::atomic`.

- **volatile**: is for special objects, on which optimized read or write operations are not allowed
- **std::atomic**: defines atomic variables, which are meant for a thread-safe reading and writing

It's so easy, but the confusion starts exactly here. The keyword `volatile` in Java and C# has the meaning of `std::atomic` in C++, i.e. `volatile` has no multithreading semantic in C++.

`volatile` is typically used in embedded programming to denote objects which can change independently of the regular program flow. One

> which can change independently of the regular program flow. One example is an object which represents an external device (memory-
>
> mapped I/O). Because these objects can change independently of the regular program flow and their value will directly be written into main memory, no optimized storing in caches takes place.

This is sufficient to synchronize two threads, so I can implement a kind of condition variable with an `std::atomic<bool>`. Therefore, let's first use a condition variable.

```
// conditionVariable.cpp

#include <condition_variable>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::mutex mutex_;
std::condition_variable condVar;

bool dataReady{false};

void waitingForWork(){
    std::cout << "Waiting " << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    mySharedWork[1] = 2;
    std::cout << "Work done " << std::endl;
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
    }
    std::cout << "Data prepared" << std::endl;
    condVar.notify_one();
}

int main(){

  std::cout << std::endl;

  std::thread t1(waitingForWork);
  std::thread t2(setDataReady);

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

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

}
```

Let me say a few words about the program. For an in-depth discussion of condition variables, read the chapter condition variables in this course.

Thread `t1` waits in line 17 for the notification of thread `t2`. Both threads use the same condition variable `condVar` and synchronize on the same mutex `mutex_`. How does the workflow run?

- thread `t2`
  - prepares the work package `mySharedWork = {1, 0, 3}`
  - set the non-atomic boolean `dataReady` to `true`
  - send its notification `condVar.notify_one`
- thread `t1`
  - waits for the notification `condVar.wait(lck, []{ return dataReady; })` while holding the lock `lck`
  - continues its work `mySharedWork[1] = 2` after getting the notification

The boolean `dataReady`, which thread `t2` sets to true and thread `t1` checks in the lambda-function `[]{ return dataReady; }`, is a kind of memory for the stateless condition variable. Condition variables may be victim to two phenomena:

1. spurious wakeup: the receiver of the message wakes up, although no notification happened
2. lost wakeup: the sender sends its notification before the receiver gets to a wait state.

Now, here's the pendant with `std::atomic<bool>`

```
// atomicCondition.cpp

#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
```

```
#include <vector>

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

void waitingForWork(){
    std::cout << "Waiting " << std::endl;
    while (!dataReady.load()){
        std::this_thread::sleep_for(std::chrono::milliseconds(5));
    }
    mySharedWork[1] = 2;
    std::cout << "Work done " << std::endl;
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady = true;
    std::cout << "Data prepared" << std::endl;
}

int main(){

  std::cout << std::endl;

  std::thread t1(waitingForWork);
  std::thread t2(setDataReady);

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

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


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

}
```

Push versus Pull Principle

Obviously I cheated a little. There is one key difference between the synchronization of the threads with a condition variable and `std::atomic<bool>`. The condition variable notifies the waiting thread (`condVar.notify()`) that it should proceed with its work. The waiting thread with `std::atomic<bool>` checks if the sender is done with its work (`dataRead = true`).

The condition variable notifies the waiting thread (push principle) while the atomic boolean repeatedly asks for the value (pull principle).

`std::atomic<bool>` and the other full or partial specializations of `std::atomic` support the bread and butter of all atomic operations: `compare_exchange_strong` and `compare_exchange_weak`.

`compare_exchange_strong` and `compare_exchange_weak`

`compare_exchange_strong` has the syntax: `bool compare_exchange_strong(T& expected, T& desired)`. Because this operation compares and exchanges its values in one atomic operation, it is often called compare and swap (CAS). This kind of operation is available in many programming languages and is the foundation of non-blocking algorithms. Of course, the behavior may vary a little. `atomicValue.compare_exchange_strong(expected, desired)` has the following behavior:

- If the atomic comparison of `atomicValue` with expected returns `true`, `atomicValue` will be set in the same atomic operation to `desired`.
- If the comparison returns `false`, `expected` will be set to `atomicValue`.

The reason the operation `compare_exchange_strong` is called strong is apparent. There is also a method `compare_exchange_weak`, although the weak version can spuriously fail. This means that although `*atomicValue == expected` holds, the weak variant returns `false`; so, you have to check the condition in a loop: `while (!atomicValue.compare_exchange_weak(expected, desired))`. The weak form exists because of performance,i.e. when called in a loop it can run faster on some platforms.

CAS operations are open for the so-called ABA problem. This means you read a value twice and each time it returns the same value A; therefore you conclude that nothing changed in between. But you overlooked that the value may have changed to B in between readings.