The remaining atomics - in contrast to std::atomic\_flag - are partial or full specialisations of the class template std::atomic. Let's start with 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. That's enough to synchronise two threads. So I can simulate [condition variables](http://modernescpp.com/index.php/condition-variables) with atomic variables.

Let's first have a look at condition variables.

// conditionVariable.cpp

#include <condition\_variable>

#include <iostream>

#include <thread>

#include <vector>

std::vector<int> mySharedWork;

std::mutex mutex\_;

std::condition\_variable condVar;

bool dataReady;

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";

}

And now the pendant with atomic booleans.

// 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() ){ **// (3)**

std::this\_thread::sleep\_for(std::chrono::milliseconds(5));

}

mySharedWork[1]= 2; **// (4)**

std::cout << "Work done " << std::endl;

}

void setDataReady(){

mySharedWork={1,0,3}; **// (1)**

dataReady= true;  **// (2)**

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";

}

What guarantees, that line 17 will be executed after the line 14? Or to say it more general, that the thread t1 will execute mySharedWork[1]= 2 (line 17) after thread t2 had executed mySharedWork={1,0,3} (line 22). Now it gets more formal.

* Line22 **(1)** *happens-before* line 23 **(2)**
* Line 14 **(3)***happens-before* line 17 **(4)**
* Line 23 **(2)** *synchronizes-with* line 14 **(3)**
* Because *happens-before*is transitive, it follows: mySharedWork={1,0,3} **(1)***happens-before* mySharedWork[1]= 2 **(4)**

In want to explicitly mention one point. Because of the condition variable condVar or the atomic dataReady, the access to the shared variable mySharedWork is synchronised. This holds although mySharedWork is not protected by a lock or itself an atomic.

Both programs produce the same result for mySharedWork.

### Push versus pull principle

Obviously, I cheated a little. There is one difference between the synchronisation of the threads with the condition variable and the atomic boolean. The condition variable notifies the waiting thread (condVar.notify()), that it should proceed with its work. But the waiting thread with the atomic boolean checks, if the sender is done with its work (dataRead= true).

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

## compare\_exchange\_strong and compare\_exchange\_weak

std::atomic<bool> and the fully or partially specialisations of std::atomic supports the bread and butter of all atomic operations: compare\_exchange\_strong. This function has the syntax: bool compare\_exchange\_strong(T& expected, T& desired). Because this operation compares and exchanges in one atomic operation a value, is often called compare\_and\_swap (CAS). This kind of operation is in a lot of programming languages available. Of course, the behaviour may differ a little.

A call of atomicValue.compare\_exchange\_strong(expected, desired) obeys the following strategy. In case the atomic comparison of atomicValue with expected returns true, the value of atomicValue is set in the same atomic operation to desired. If the comparison returns false, expected will be set to atomicValue. The reason why the operation compare\_exchange\_strong is called strong is simple. There is a method compare\_exchange\_weak. This weak version can spuriously fail. That mean, 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 reason for the weak form is performance. On some platforms, the weak is faster than the strong variant.