

Condition Variables

Let's talk about condition variables that enable threads to wait until a particular condition occurs.

Condition variables enable threads to be synchronized via messages. They need the header `<condition_ variable>`. One thread acts as a sender, and the other as a receiver of the message. The receiver waits for the notification of the sender. Typical use cases for condition variables are producer-consumer workflows. A condition variable can be the sender but also the receiver of the message.


Method	Description
<code>cv.notify_one()</code>	Notifies a waiting thread.
<code>cv.notify_all()</code>	Notifies all waiting threads.
<code>cv.wait(lock, ...)</code>	Waits for the notification while holding a <code>std::unique_lock</code> .
<code>cv.wait_for(lock, relTime, ...)</code>	Waits for a time duration for the notification while holding a <code>std::unique_lock</code> .
<code>cv.wait_until(lock, absTime, ...)</code>	Waits until a time for the notification while holding a <code>std::unique_lock</code> .

Sender and receiver need a lock. In case of the sender a `std::lock_guard` is sufficient because it only once calls `lock` and `unlock`. In the case of the receiver a `std::unique_lock` is necessary because it typically often locks and

unlocks its mutex.

```
//...
#include <condition_variable>
//...
std::mutex mutex_;
std::condition_variable condVar;
bool dataReady= false;
void doTheWork(){
    std::cout << "Processing shared data." << std::endl;
}
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}
void setDataReady(){
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << "Sender: Data is ready." << std::endl;
    condVar.notify_one();
}

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



```
rainer : bash - Konsole
Datei  Bearbeiten  Ansicht  Lesezeichen  >
rainer@icho:~> conditionVariable

Worker: Waiting for work.
Sender: Data is ready.
Processing shared data.
Work done.

rainer@icho:~> █
```

Using a condition variable may sound easy, but there are two critical issues.

⚠ Protection against spurious wakeup

The first critical issue is that a thread can wake up without any notification.

To protect itself against spurious wakeup, the wait call of the condition variable should use an additional predicate. The predicate ensures that the notification is indeed from the sender. I use the `lambda function []{ return dataReady; }` as the predicate. `dataReady` is set to true by the sender.

⚠ **Protection against lost wakeup**

To protect itself against lost wakeup, the wait call of the condition variable should use an additional predicate. The predicate ensures that the notification of the sender is not lost. The notification is lost if the sender notifies the receiver before the receiver is waiting. Therefore the receiver waits forever. The receiver now checks at first its `predicate: [] { return dataReady; }`.