# Condition Variables

This lesson explains condition variable such as wait s and their usage in C++ for multithreading purposes.

WE'LL COVER THE FOLLOWING

- The Wait Workflow

Condition variables enable threads to be synchronized via messages. They need the `<condition_variable>` header, one thread to act as a sender, and the other as the receiver of the message; the receiver waits for the notification from the sender. Typical use cases for condition variables are sender-receiver or producer-consumer workflows.

A condition variable can be the sender but also the receiver of the message.

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

The subtle difference between `cv.notify_one` and `cv.notify_all` is that `cv.notify_all` will notify all waiting threads. In contrast, `cv.notify_one` will notify only one of the waiting threads while the other threads remain in the wait state. Before we cover the gory details of condition variables - which are the three dots in the wait operations - here is an example.

```
// conditionVariable.cpp

#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

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();
}

int main(){

  std::cout << std::endl;

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

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

  std::cout << std::endl;

}
```

The program has two child threads: `t1` and `t2`. They get their work package `waitingForWork` and `setDataRead` in lines 38 and 39. `setDataReady` notifies - using the condition variable `condVar` - that it is done with the preparation of the work: `condVar.notify_one()`. While holding the lock, thread `t1` waits for its notification: `condVar.wait(lck,[]{ return dataReady; })`. Meanwhile, the sender and receiver need a lock. In the case of the sender a `std::lock_guard` is sufficient, because it calls to lock and unlock only once. In the case of the receiver, a `std::unique_lock` is necessary because it frequently locks and unlocks its mutex. The waiting thread has quite a complicated workflow.

## The Wait Workflow

If it is the first time `wait` is invoked, the following steps will happen.

- The call to `wait` locks the mutex and checks if the predicate `[]{ return dataReady; }` evaluates to true.
  - If true, the condition variable unlocks the mutex and continues.
  - If false, the condition variable unlocks the mutex and puts itself back in the wait state.

Subsequent `wait` calls behave differently:

- The waiting thread gets a notification. It locks the mutex and checks if the predicate `[]{ return dataReady; }` evaluates to true.
  - If true, the condition variable unlocks the mutex and continues.
  - If false, the condition variable unlocks the mutex and puts itself back in the wait state.

Maybe you are wondering why you need a predicate for the `wait` call when you can invoke wait without a predicate? Let's try it out.

```
// conditionVariableBlock.cpp

#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;
```

```
std::mutex mutex_;
std::condition_variable condVar;

bool dataReady{false};


void waitingForWork(){

    std::cout << "Worker: Waiting for work." << std::endl;

    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    // do the work
    std::cout << "Work done." << std::endl;

}

void setDataReady(){

    std::cout << "Sender: Data is ready."  << std::endl;
    condVar.notify_one();

}

int main(){

  std::cout << std::endl;

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

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

  std::cout << std::endl;

}
```

The first invocation of the program seems to work fine. The second invocation locks because the notification call (line 28) happens before thread `t2` (line 37) enters the waiting state (line 19).

Now it is clear. The predicate is a kind of memory for the stateless condition variable; therefore, the wait call always checks the predicate at first. Condition variables are victim to two known phenomena: lost wakeup and spurious wakeup. We will discuss these phenomena in the next lesson.