**Condition Variable:**

The condition\_variable class is a synchronization primitive used with a [std::mutex](https://en.cppreference.com/w/cpp/thread/mutex) to block one or more threads until another thread both modifies a shared variable (the *condition*) and notifies the condition\_variable.

The thread that intends to modify the shared variable must:

1. Acquire a std::mutex (typically via [std::lock\_guard](https://en.cppreference.com/w/cpp/thread/lock_guard))
2. Modify the shared variable while the lock is owned
3. Call [notify\_one](https://en.cppreference.com/w/cpp/thread/condition_variable/notify_one) or [notify\_all](https://en.cppreference.com/w/cpp/thread/condition_variable/notify_all) on the std::condition\_variable (can be done after releasing the lock)

Even if the shared variable is atomic, it must be modified while owning the mutex to correctly publish the modification to the waiting thread.

Any thread that intends to wait on a std::condition\_variable must:

1. Acquire a [std::unique\_lock](https://en.cppreference.com/w/cpp/thread/unique_lock)<[std::mutex](https://en.cppreference.com/w/cpp/thread/mutex)> on the mutex used to protect the shared variable
2. Do one of the following:
3. Check the condition in case it was already updated and notified.
4. Call [wait](https://en.cppreference.com/w/cpp/thread/condition_variable/wait), [wait\_for](https://en.cppreference.com/w/cpp/thread/condition_variable/wait_for), or [wait\_until](https://en.cppreference.com/w/cpp/thread/condition_variable/wait_until) on the std::condition\_variable (atomically releases the mutex and suspends thread execution until the condition variable is notified, a timeout expires, or a [spurious wakeup](https://en.wikipedia.org/wiki/Spurious_wakeup) occurs, then atomically acquires the mutex before returning)
5. Check the condition and resume waiting if not satisfied.

or:

1. Use the predicated overload of [wait](https://en.cppreference.com/w/cpp/thread/condition_variable/wait), [wait\_for](https://en.cppreference.com/w/cpp/thread/condition_variable/wait_for), and [wait\_until](https://en.cppreference.com/w/cpp/thread/condition_variable/wait_until), which performs the same three steps

std::condition\_variable works only with [std::unique\_lock](https://en.cppreference.com/w/cpp/thread/unique_lock)<[std::mutex](https://en.cppreference.com/w/cpp/thread/mutex)>, which allows for maximal efficiency on some platforms. [std::condition\_variable\_any](https://en.cppreference.com/w/cpp/thread/condition_variable_any) provides a condition variable that works with any [*BasicLockable*](https://en.cppreference.com/w/cpp/named_req/BasicLockable) object, such as [std::shared\_lock](https://en.cppreference.com/w/cpp/thread/shared_lock).

Condition variables permit concurrent invocation of the [wait](https://en.cppreference.com/w/cpp/thread/condition_variable/wait), [wait\_for](https://en.cppreference.com/w/cpp/thread/condition_variable/wait_for), [wait\_until](https://en.cppreference.com/w/cpp/thread/condition_variable/wait_until), [notify\_one](https://en.cppreference.com/w/cpp/thread/condition_variable/notify_one) and [notify\_all](https://en.cppreference.com/w/cpp/thread/condition_variable/notify_all) member functions.

The class std::condition\_variable is a [*StandardLayoutType*](https://en.cppreference.com/w/cpp/named_req/StandardLayoutType). It is not [*CopyConstructible*](https://en.cppreference.com/w/cpp/named_req/CopyConstructible), *[MoveConstructible](https://en.cppreference.com/w/cpp/named_req/MoveConstructible" \o "cpp/named req/MoveConstructible)*, *[CopyAssignable](https://en.cppreference.com/w/cpp/named_req/CopyAssignable" \o "cpp/named req/CopyAssignable)*, or *[MoveAssignable](https://en.cppreference.com/w/cpp/named_req/MoveAssignable" \o "cpp/named req/MoveAssignable)*.

**Sample Code:**

#include <iostream>

#include <string>

#include <thread>

#include <mutex>

#include <condition\_variable>

[std::mutex](https://en.cppreference.com/w/cpp/thread/mutex) m;

std::condition\_variable cv;

[std::string](https://en.cppreference.com/w/cpp/string/basic_string) data;

bool ready = false;

bool processed = false;

void worker\_thread()

{

// Wait until main() sends data

[std::unique\_lock](https://en.cppreference.com/w/cpp/thread/unique_lock) lk(m);

cv.wait(lk, []{return ready;});

// after the wait, we own the lock.

[std::cout](https://en.cppreference.com/w/cpp/io/cout) << "Worker thread is processing data**\n**";

data += " after processing";

// Send data back to main()

processed = true;

[std::cout](https://en.cppreference.com/w/cpp/io/cout) << "Worker thread signals data processing completed**\n**";

// Manual unlocking is done before notifying, to avoid waking up

// the waiting thread only to block again (see notify\_one for details)

lk.unlock();

cv.notify\_one();

}

int main()

{

[std::thread](https://en.cppreference.com/w/cpp/thread/thread) worker(worker\_thread);

data = "Example data";

// send data to the worker thread

{

[std::lock\_guard](https://en.cppreference.com/w/cpp/thread/lock_guard) lk(m);

ready = true;

[std::cout](https://en.cppreference.com/w/cpp/io/cout) << "main() signals data ready for processing**\n**";

}

cv.notify\_one();

// wait for the worker

{

[std::unique\_lock](https://en.cppreference.com/w/cpp/thread/unique_lock) lk(m);

cv.wait(lk, []{return processed;});

}

[std::cout](https://en.cppreference.com/w/cpp/io/cout) << "Back in main(), data = " << data << '**\n**';

worker.join();

}

Output:

main() signals data ready for processing

Worker thread is processing data

Worker thread signals data processing completed

Back in main(), data = Example data after processing

**Print Even-Odd from separate threads:**

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

std::mutex mu;

std::condition\_variable cond;

int count = 1;

bool even = false;

bool odd = true;

void PrintOdd()

{

for (; count < 100;)

{

std::unique\_lock<std::mutex> locker(mu);

//std::cout << "Start ODD, count:" << count << std::endl;

cond.wait(locker, []() { return odd; });

std::cout << "From Odd: " << count << std::endl;

count++;

even = true;

odd = false;

locker.unlock();

cond.notify\_one();

}

}

void PrintEven()

{

for (; count < 100;)

{

std::unique\_lock<std::mutex> locker(mu);

//std::cout << "Start EVEN count:" << count << std::endl;

cond.wait(locker, []() { return even; });

std::cout << "From Even: " << count << std::endl;

count++;

even = false;

odd = true;

locker.unlock();

cond.notify\_one();

}

}

int main()

{

std::thread t1(PrintOdd);

std::thread t2(PrintEven);

t1.join();

t2.join();

return 0;

}

**Produce Consumer sharing data in a queue:**

Workflow:

Producer produces 1, 2, … 10 and signals consumer to print.

Consumer prints 1, 2, … 10 and signals the producer to generate

Producer produces 11, 12 … 20 and signals consumer to print.

Consumer prints 1, 2, … 10 and signals the producer to generate

…

…

#include <iostream>

#include <string>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <queue>

std::mutex m;

std::condition\_variable cv;

std::string data;

bool produced = false;

bool consumed = true;

std::queue<int> q1;

int val = 1;

int limit = 100;

int interval = 10;

void producer()

{

for (; val < limit;)

{

// Wait until main() sends data

std::unique\_lock<std::mutex> lk(m);

cv.wait(lk, [] { return consumed; });

// after the wait, we own the lock.

std::cout << "Producer thread is generating data\n";

for (int i = 0; i < interval; i++)

{

q1.push(val);

val++;

}

// Send data back to main()

produced = true;

consumed = false;

std::cout << "Producer thread signals data generation completed\n";

// Manual unlocking is done before notifying, to avoid waking up

// the waiting thread only to block again (see notify\_one for details)

lk.unlock();

cv.notify\_one();

}

}

void consumer()

{

while (true)

{

// Wait until main() sends data

std::unique\_lock<std::mutex> lk(m);

cv.wait(lk, [] {return produced; });

// after the wait, we own the lock.

std::cout << "--Consumer thread is consuming data\n";

while (!q1.empty())

{

int front = q1.front();

std::cout << front << " ";

q1.pop();

}

std::cout << "\n";

// Send data back to main()

consumed = true;

produced = false;

std::cout << "--Consumer thread signals data processing completed\n";

// Manual unlocking is done before notifying, to avoid waking up

// the waiting thread only to block again (see notify\_one for details)

lk.unlock();

cv.notify\_one();

}

}

int main()

{

std::thread tProducer(producer);

std::thread tConsumer(consumer);

tProducer.join();

tConsumer.join();

}