Condition Variable:

The condition_variable class is a synchronization primitive used with a std::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)
2. Modify the shared variable while the lock is owned
3. Call notify_one or 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<std::mutex> on the mutex used to protect the shared variable
2. Do one of the following:
    1. Check the condition in case it was already updated and notified.
    2. Call wait, wait_for, or 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 occurs, then atomically acquires the mutex before returning)
    3. Check the condition and resume waiting if not satisfied.
       or:
        1. Use the predicated overload of wait, wait_for, and wait_until, which performs the same three steps

std::condition_variable works only with std::unique_lock<std::mutex>, which allows for maximal efficiency on some platforms. std::condition_variable_any provides a condition variable that works with any *BasicLockable* object, such as std::shared_lock.

Condition variables permit concurrent invocation of the wait, wait_for, wait_until, notify_one and notify_all member functions.

The class std::condition_variable is a *StandardLayoutType*. It is not *CopyConstructible*, *MoveConstructible*, *CopyAssignable*, or *MoveAssignable*.

Sample Code:

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

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;
```

```cpp
void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::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 worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::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:

```cpp
#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)
```

```cpp
        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();
}
```