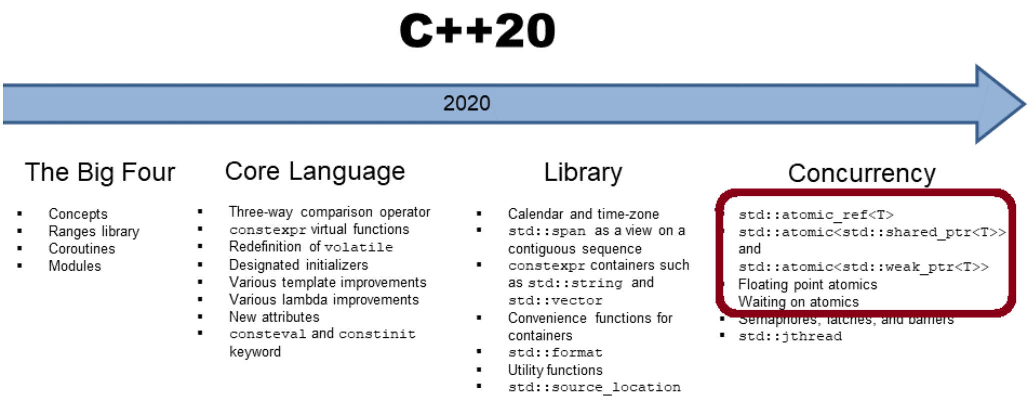
Atomic References

Atomics receives a few important extensions in C++20. Today, I start with the new data type `std::atomic_ref`.



The type `std::atomic_ref` applies atomic operations to its referenced object.

`std::atomic_ref`

Concurrent writing and reading using a `std::atomic_ref` is no data race. The lifetime of the referenced object must exceed the lifetime of the `std::atomic_ref`. Accessing a subobject of the referenced object with a `std::atomic_ref` is not well-defined.

Motivation

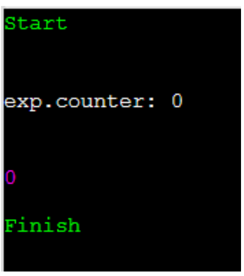
You may think that using a reference inside an atomic would do the job. Unfortunately not.

In the following program, I have a class `ExpensiveToCopy`, which includes a `counter`. The `counter` is concurrently incremented by a few threads. Consequently, `counter` has to be protected.

```cpp
// atomicReference.cpp

#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {             // (6)

    std::random_device seed;        // initial seed
    std::mt19937 engine(seed());    // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
}

void count(ExpensiveToCopy& exp) {                          // (2)

    std::vector<std::thread> v;
    std::atomic<int> counter{exp.counter};                  // (3)

    for (int n = 0; n < 10; ++n) {                          // (4)
        v.emplace_back([&counter] {
            auto randomNumber = getRandom(100, 200);        // (5)
            for (int i = 0; i < randomNumber; ++i) { ++counter; }
        });
    }

    for (auto& t : v) t.join();

}

int main() {

    std::cout << std::endl;

    ExpensiveToCopy exp;                 // (1)
    count(exp);
    std::cout << "exp.counter: " << exp.counter << '\n';

    std::cout << std::endl;

}
```

`exp` (1) is the expensive-to-copy object. For performance reasons, the function `count` (2) takes `exp` by reference. `count` initializes the `std::atomic<int>` with `exp.counter` (3). The following lines create 10 threads (4), each performing the lambda expression, which takes `counter` by reference. The lambda expression gets a random number between 100 and 200 (5) and increments the counter exactly as often. The function `getRandom` (6) start with an initial seed and creates via the random number generator [Mersenne Twister](#) a uniform distributed number.

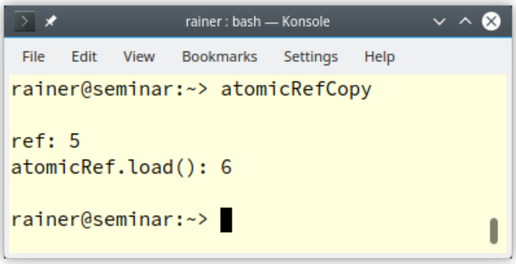
In the end, the `exp.counter` (7) should have an approximate value of 1500 because of the ten threads increments on average 150 times. Executing the program on the [Wandbox](#) online compiler gives me a surprising result.



The counter is 0. What is happening? The issue is in line (3). The initialization in the expression `std::atomic<int> counter{exp.counter}` creates a copy. The following small program exemplifies the issue.

```
// atomicRefCopy.cpp

#include <atomic>
#include <iostream>

int main() {

    std::cout << std::endl;

    int val{5};
    int& ref = val;                    // (2)
    std::atomic<int> atomicRef(ref);
    ++atomicRef;                       // (1)
    std::cout << "ref: " << ref << std::endl;
    std::cout << "atomicRef.load(): " << atomicRef.load() << std::endl;

    std::cout << std::endl;

}
```

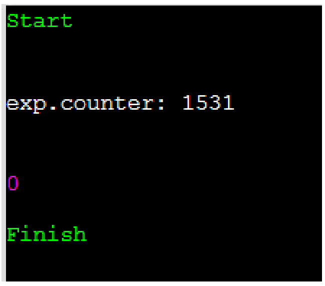
The increment operation (1) does not address the reference `ref` (2). The value of `ref` is not changed.



Replacing the `std::atomic<int> counter{exp.counter}` with `std::atomic_ref<int> counter{exp.counter}` solves the issue:

```cpp
// atomicReference.cpp

#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {

    std::random_device seed;        // initial randomness
    std::mt19937 engine(seed());    // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
}

void count(ExpensiveToCopy& exp) {

    std::vector<std::thread> v;
    std::atomic_ref<int> counter{exp.counter};

    for (int n = 0; n < 10; ++n) {
        v.emplace_back([&counter] {
            auto randomNumber = getRandom(100, 200);
            for (int i = 0; i < randomNumber; ++i) { ++counter; }
        });
    }

    for (auto& t : v) t.join();

}

int main() {

    std::cout << std::endl;

    ExpensiveToCopy exp;
    count(exp);
    std::cout << "exp.counter: " << exp.counter << '\n';

    std::cout << std::endl;

}
```

Now, the value of `counter` is as expected:



To be Atomic or Not to be Atomic

You may ask me why I didn't make the counter atomic in the first place:

```cpp
struct ExpensiveToCopy {
    std::atomic<int> counter{};
};
```

Of course, this is a valid approach, but this approach has a big downside. Each access of the counter is synchronized, and synchronization is not for free. On the contrary, using a `std::atomic_ref<int> counter` lets you explicitly control when you need atomic access to the counter. Maybe, most of the time, you only want to read the value of the counter. Consequently, defining it as an atomic is pessimization.

Let me conclude my post with a few more details to the class template `std::atomic_ref`.

Specializations of `std::atomic_ref`

You can specialize `std::atomic_ref` for user-defined type, use partially specializations for pointer types or full specializations for arithmetic types such as integral or floating-point types.

Primary Template

The primary template `std::atomic_ref` can be instantiated with a trivially copyable type T. Trivially copyable types are either scalar types (arithmetic types, `enum`'s, pointers, member pointers, or `std::nullptr_t`'s), or trivially copyable classes and arrays of scalar types

Partial Specializations for Pointer Types

The standard provides partial specializations for a pointer type: `std::atomic_ref<t*>`.

Specializations for Arithmetic Types

The standard provides specialization for the integral and floating-point types: `std::atomic_ref<arithmetic type>`.

- Character types: `char, char8_t` (C++20), char16_t, char32_t, and `wchar_t`
- Standard signed integer types: `signed char, short, int, long,` and long long
- Standard unsigned integer types: `unsigned char, unsigned short, unsigned int, unsigned long`, and `unsigned long long`
- Additional integer types, defined in the header `<cstdint>`
- Standard floating-point types: `float`, `double`, and `long double`

All Atomic Operations

First, here is the list of all operations on `std::atomic_ref`.

Function	Description
is_lock_free	Checks if the atomic_ref object is lock-free.
load	Atomically returns the value of the referenced object.
store	Atomically replaces the value of the referenced object with a non-atomic.
exchange	Atomically replaces the value of the referenced object with the new value.
compare_exchange_strong	Atomically compares and eventually exchanges the value of the referenced object.
compare_exchange_weak	
fetch_add, +=	Atomically adds(subtracts) the value to(from) the referenced object.
fetch_sub, -=	
fetch_or, |=	Atomically performs bitwise (OR, AND, and XOR) operation on the referenced object.
fetch_and, &=	
fetch_xor, ^=	
++, --	Increments or decrements (pre- and post-increment) the referenced object.
notify_one	Unblocks one atomic wait operation.
notify_all	Unblocks all atomic wait operations.
wait	Blocks until it is notified.

The composite assignment operators (`+=, -=, |=, &=`, or `^=`) return the new value; the `fetch` variations return the old value. The `compare_exchange_strong` and `compare_exchange_weak` perform an atomic `exchange` if equal and an atomic `load` if not. They return `true` in the success case, otherwise `false`. Each function supports an additional memory-ordering argument. The default is sequential consistency.

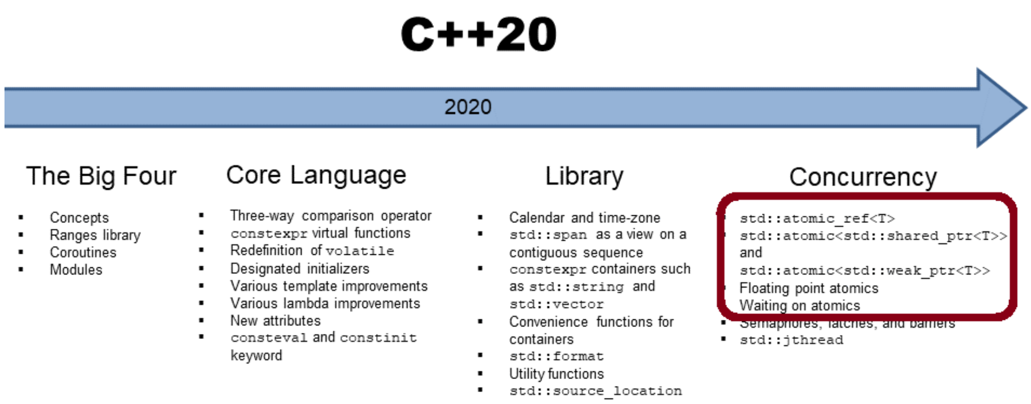
Of course, not all operations are available on all types referenced by `std::atomic_ref`. The table shows the list of all atomic operations depending on the type referenced by `std::atomic_ref`.

Function	atomic_ref<T>	atomic_ref<integral>	atomic_ref<floating>	atomic_ref<T*>
is_lock_free	yes	yes	yes	yes
load	yes	yes	yes	yes
store	yes	yes	yes	yes
exchange	yes	yes	yes	yes
compare_exchange_strong	yes	yes	yes	yes
compare_exchange_weak	yes	yes	yes	yes
fetch_add, +=		yes	yes	yes
fetch_sub, -=		yes	yes	yes
fetch_or, |=		yes		
fetch_and, &=		yes		
fetch_xor, ^=		yes		
++, --		yes		yes
notify_one	yes	yes	yes	yes
notify_all	yes	yes	yes	yes
wait	yes	yes	yes	yes

When you study the last two tables carefully, you notice that you can use`std::atomic_ref` to synchronize threads.

Synchronization with Atomics

Sender/receiver workflows are quite common for threads. In such a workflow, the receiver is waiting for the sender's notification before it continues to work. There are various ways to implement these workflows. With C++11, you can use condition variables or promise/future pairs; with C++20, you can use atomics.



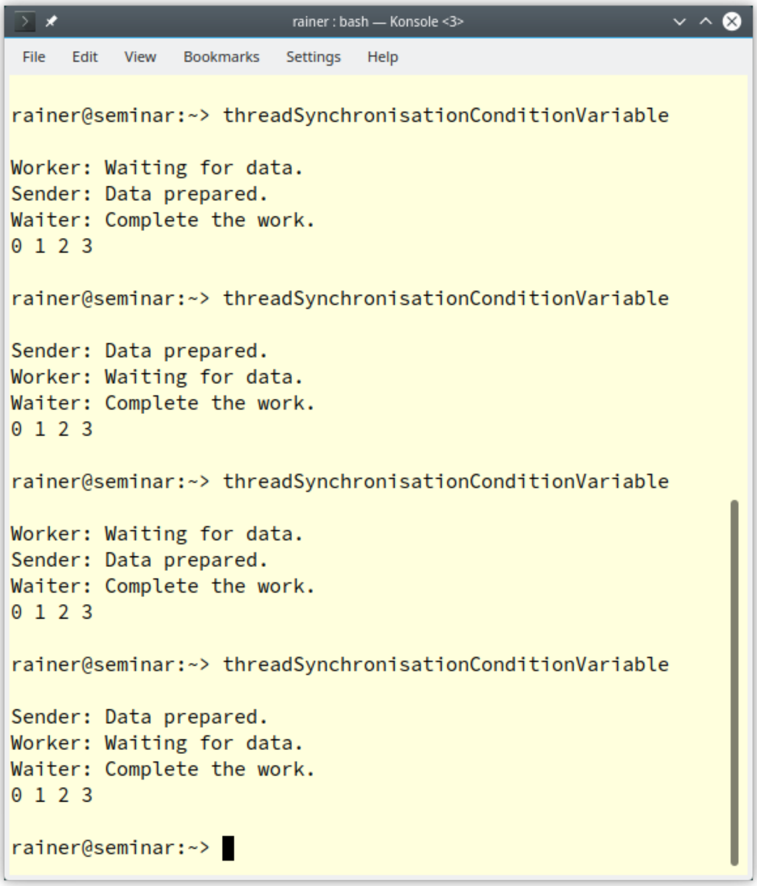
There are various ways to synchronize threads. Each way has its pros and cons. Consequently, I want to compare them. I assume you don't know the details to condition variables or promise and futures. Therefore, I give a short refresher.

Condition Variables

A condition variable can fulfill the role of a sender or a receiver. As a sender, it can notify one or more receivers.

```cpp
// threadSynchronisationConditionVariable.cpp

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

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

std::vector<int> myVec{};

void prepareWork() {                                           // (1)

    {
        std::lock_guard<std::mutex> lck(mutex_);
        myVec.insert(myVec.end(), {0, 1, 0, 3});               // (3)
    }
    std::cout << "Sender: Data prepared."  << std::endl;
    condVar.notify_one();
}

void completeWork() {                                           // (2)

    std::cout << "Worker: Waiting for data." << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, [] { return not myVec.empty(); });
    myVec[2] = 2;                                               // (4)
    std::cout << "Waiter: Complete the work." << std::endl;
    for (auto i: myVec) std::cout << i << " ";
    std::cout << std::endl;

}

int main() {

    std::cout << std::endl;

    std::thread t1(prepareWork);
    std::thread t2(completeWork);

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

    std::cout << std::endl;

}
```

The program has two child threads:`t1` and `t2`. They get their payload `prepareWork` and `completeWork` in lines (1) and (3). The function `prepareWork` notifies that it is done with the preparation of the work:`condVar.notify_one()`. While holding the lock, the thread `t2` is waiting for its notification: `condVar.wait(lck, []{ return not myVec.empty(); })`. The waiting thread always performs the same steps. When it is waked up, it checks the predicate while holding the lock (`[]{ return not myVec.empty();`). If the predicate does not hold, it puts itself back to sleep. If the predicate holds, it continues with its work. In the concrete workflow, the sending thread puts the initial values into the `std::vector`(3), which the receiving thread completes (4).



Condition variables have many inherent issues. For example, the receiver could be awakened without notification or could lose the notification. The first issue is known as spurious wakeup and the second as lost wakeup. The predicate protects against both flaws. The notification would be lost when the sender sends its notification before the receiver is in the wait state and does not use a predicate. Consequently, the receiver waits for something that never happens. This is a deadlock. When you study the output of the program, you see, that each second run would cause a deadlock if I would not use a predicate. Of course, it is possible to use condition variables without a predicate.

If you want to know the details of the sender/receiver workflow and the traps of condition variables, read my previous posts "C++ Core Guidelines: Be Aware of the Traps of Condition Variables".

When you only need a one-time notification such as in the previous program, promises and futures are a better choice than condition variables. Promise and futures cannot be victims of spurious or lost wakeups.

Promises and Futures

A promise can send a value, an exception, or a notification to its associated future. Let me use a promise and a future to refactor the previous workflow. Here is the same workflow using a promise/future pair.

```cpp
// threadSynchronisationPromiseFuture.cpp

#include <iostream>
#include <future>
#include <thread>
#include <vector>

std::vector<int> myVec{};

void prepareWork(std::promise<void> prom) {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << std::endl;
    prom.set_value();                                     // (1)

}

void completeWork(std::future<void> fut){

    std::cout << "Worker: Waiting for data." << std::endl;
    fut.wait();                                           // (2)
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << std::endl;
    for (auto i: myVec) std::cout << i << " ";
    std::cout << std::endl;

}

int main() {

    std::cout << std::endl;

    std::promise<void> sendNotification;
    auto waitForNotification = sendNotification.get_future();

    std::thread t1(prepareWork, std::move(sendNotification));
    std::thread t2(completeWork, std::move(waitForNotification));

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

    std::cout << std::endl;

}
```

When you study the workflow, you recognize, that the synchronization is reduced to its essential parts:`prom.set_value()` (1) and `fut.wait() (2)`. There is neither a need to use locks or mutexes, nor is there a need to use a predicate to protect against spurious or lost wakeups. I skip the screen-shot to this run because it is essentially the same such in the case of the previous run with condition variables.

There is only one downside to using promises and futures: they can only be used once. Here are my previous posts to [promises and futures](), often just called tasks.

If you want to communicate more than once, you have to use condition variables or atomics.

std::atomic_flag

`std::atomic_flag` in C++11 has a simple interface. It's member function `clear` enables you to set its value to `false`, with `test_and_set` to `true`. In case you use `test_and_set` you get the old value back. `ATOMIC_FLAG_INIT` enables it to initialize the `std::atomic_flag` to `false`. `std::atomic_flag` has two very interesting properties.

`std::atomic_flag` is

- the only lock-free atomic.
- the building block for higher thread abstractions.

The remaining more powerful atomics can provide their functionality by using a mutex. That is according to the C++ standard. So these atomics have a member function `is_lock_free` .On the popular platforms, I always get the answer `false`. But you should be aware of that. Here are more details on the capabilities of [`std::atomic_flag`]() in C++11.

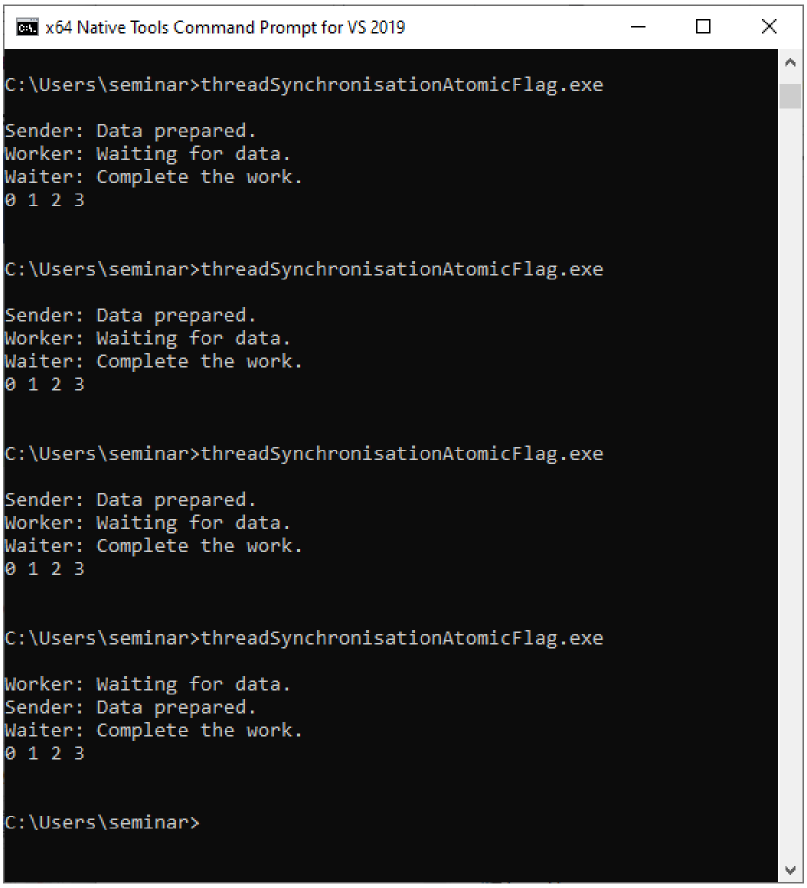
Now, I jump directly from C++11 to C++20. With C++20, `std::atomic_flag atomicFlag` support new member functions: `atomicFlag.wait()`, `atomicFlag.notify_one()`, and `atomicFlag.notify_all()`. The member functions `notify_one` or `notify_all` notify one or all of the waiting atomic flags.`atomicFlag.wait(boo)` needs a boolean `boo`. The call `atomicFlag.wait(boo)` blocks until the next notification or spurious wakeup. It checks then if the value of `atomicFlag` is equal to `boo` and unblocks if not. The value `boo` serves as a kind of predicate.

Additionally to C++11, default-construction of a `std::atomic_flag` sets it in its `false` state and you can ask for the value of the `std::atomic flag` via `atomicFlag.test()`. With this knowledge, it's quite easy to refactor to previous programs using a `std::atomic_flag`.

```
// threadSynchronisationAtomicFlag.cpp

#include <iostream>
#include <atomic>
#include <thread>
#include <vector>

std::vector<int> myVec{};

std::atomic_flag atomicFlag{};

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << std::endl;
    atomicFlag.test_and_set();                              // (1)
    atomicFlag.notify_one();

}

void completeWork() {

    std::cout << "Worker: Waiting for data." << std::endl;
    atomicFlag.wait(false);                                 // (2)
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << std::endl;
    for (auto i: myVec) std::cout << i << " ";
    std::cout << std::endl;

}

int main() {

    std::cout << std::endl;

    std::thread t1(prepareWork);
    std::thread t2(completeWork);

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

    std::cout << std::endl;

}
```

The thread preparing the work (1) sets the `atomicFlag` to `true` and sends the notification. The thread completing the work waits for the notification. It is only unblocked if `atomicFlag` is equal to `true`.

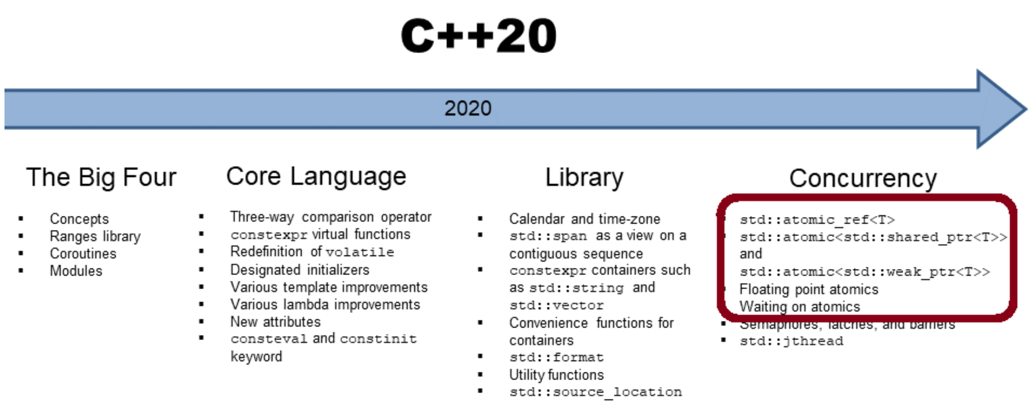
Here are a few runs of the program with the Microsoft Compiler.



I'm not sure if I would use a future/promise pair or a`std::atomic_flag` for such a simple thread synchronization workflow. Both are thread-safe by design and require no protection mechanism so far. Promise and promise are easier to use but `std::atomic_flag` is probably faster. I'm only sure that I would not use a condition variable if possible.

Performance Comparison of Condition Variables and Atomics

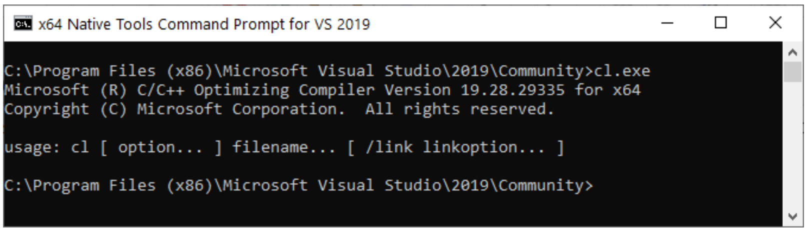
After the introduction to `std::atomic_flag` in my last post Synchronisation with Atomics in C++20, I want to dive deeper. Today, I create a ping-pong game using condition variables, `std::atomic_flag`, and `std::atomic<bool>`. Let's play.



The key question I want to answer in this post is the following: What is the fastest way to synchronize threads in C++20? I use in this post three different data types: `std::condition_variable`, `std::atomic_flag`, and `std::atomic<bool>`.

To get comparable numbers, I implement a ping-pong game. One thread executes a`ping` function and the other thread a `pong` function. For simplicity reasons, I call the thread executing the`ping` function the ping thread and the other thread the pong thread. The ping thread waits for the notification of the pong threads and sends the notification back to the pong thread. The game stops after 1,000,000 ball changes. I perform each game five times to get comparable performance numbers.

I made my performance test with the brand new Visual Studio compiler because it already supports synchronization with atomics. Additionally, I compiled the examples with maximum optimization (`/Ox`).



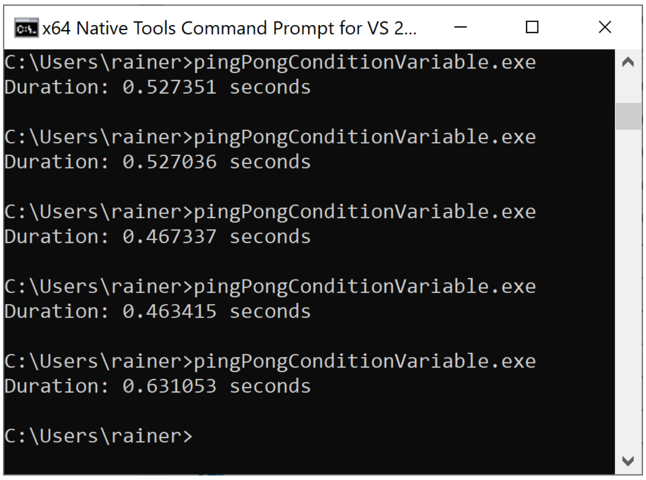
Let me start with the C++11.

Condition Variables

```cpp
// pingPongConditionVariable.cpp

#include <condition_variable>
#include <iostream>
#include <atomic>
#include <thread>

bool dataReady{false};

std::mutex mutex_;
std::condition_variable condVar1;          // (1)
std::condition_variable condVar2;          // (2)

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {

    while(counter <= countlimit) {
        {
            std::unique_lock<std::mutex> lck(mutex_);
            condVar1.wait(lck, []{return dataReady == false;});
            dataReady = true;
        }
        ++counter;
        condVar2.notify_one();              // (3)
  }
}

void pong() {

    while(counter < countlimit) {
        {
            std::unique_lock<std::mutex> lck(mutex_);
            condVar2.wait(lck, []{return dataReady == true;});
            dataReady = false;
        }
        condVar1.notify_one();            // (3)
  }

}

int main(){

    auto start = std::chrono::system_clock::now();

    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << std::endl;

}
```

I use two condition variables in the program:`condVar1` and `condVar2` (line 1 and 2). The ping thread wait for the notification of `condVar1` and sends its notification with `condVar2`. `dataReady` protects against spurious and lost wakeups (see "[C++ Core Guidelines: Be Aware of the Traps of Condition Variables]()"). The ping-pong game ends when `counter` reaches the `countlimit`. The `nofication_one` calls (lines 3) and the counter are thread-safe and are, therefore, outside the critical region.

Here are the numbers:



The average execution time is 0.52 seconds.

Porting this play to `std::atomic_flags`'s in C++20 is straightforward.

`std::atomic_flag`

Here is the play using two atomic flags.

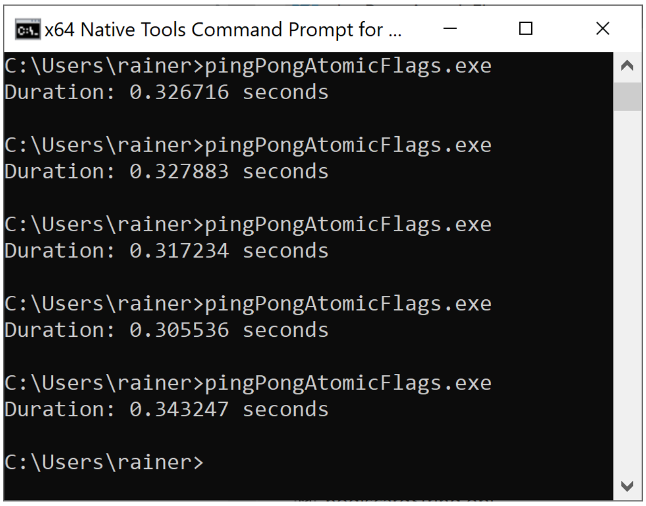
Two Atomic Flags

In the following program, I replace the waiting on the condition variable with the waiting on the atomic flag and the notification of the condition variable with the setting of the atomic flag followed by the notification.

```cpp
// pingPongAtomicFlags.cpp

#include <iostream>
#include <atomic>
#include <thread>

std::atomic_flag condAtomicFlag1{};
std::atomic_flag condAtomicFlag2{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        condAtomicFlag1.wait(false);               // (1)
        condAtomicFlag1.clear();                   // (2)

        ++counter;

        condAtomicFlag2.test_and_set();           // (4)
        condAtomicFlag2.notify_one();             // (3)
    }
}

void pong() {
    while(counter < countlimit) {
        condAtomicFlag2.wait(false);
        condAtomicFlag2.clear();

        condAtomicFlag1.test_and_set();
        condAtomicFlag1.notify_one();
    }
}

int main() {

     auto start = std::chrono::system_clock::now();

    condAtomicFlag1.test_and_set();                    // (5)
    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << std::endl;

}
```

A call `condAtomicFlag1.wait(false)` (1) blocks, if the value of the atomic flag is `false`. On the contrary, it returns if `condAtomicFlag1` has the value `true`. The boolean value serves as a kind of predicate and must, therefore, set back to `false` (2). Before the notification (3) is sent to the pong thread, `condAtomicFlag1` is set to `true` (4). The initial setting of `condAtomicFlag1` to `true` (5) starts the game.

Thanks to `std::atomic_flag` the game ends earlier.



On average, a game takes 0.32 seconds.

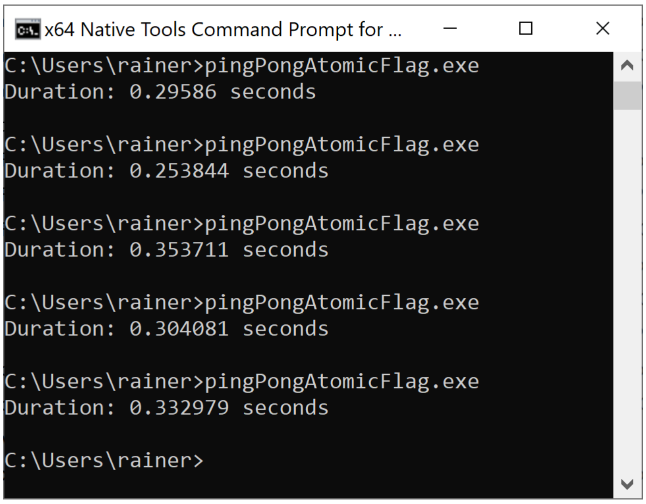
When you analyze the program, you may recognize, that one atomics flag is sufficient for the play.

One Atomic Flag

Using one atomic flag makes the play easier to understand.

```
// pingPongAtomicFlag.cpp

#include <iostream>
#include <atomic>
#include <thread>

std::atomic_flag condAtomicFlag{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        condAtomicFlag.wait(true);
        condAtomicFlag.test_and_set();

        ++counter;

        condAtomicFlag.notify_one();
    }
}

void pong() {
    while(counter < countlimit) {
        condAtomicFlag.wait(false);
        condAtomicFlag.clear();
        condAtomicFlag.notify_one();
    }
}

int main() {

     auto start = std::chrono::system_clock::now();


    condAtomicFlag.test_and_set();
    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << std::endl;

}
```

In this case, the ping thread blocks on `true` but the pong thread blocks on `false`. From the performance perspective, using one or two atomic flags makes no difference.



The average execution time is 0.31 seconds.

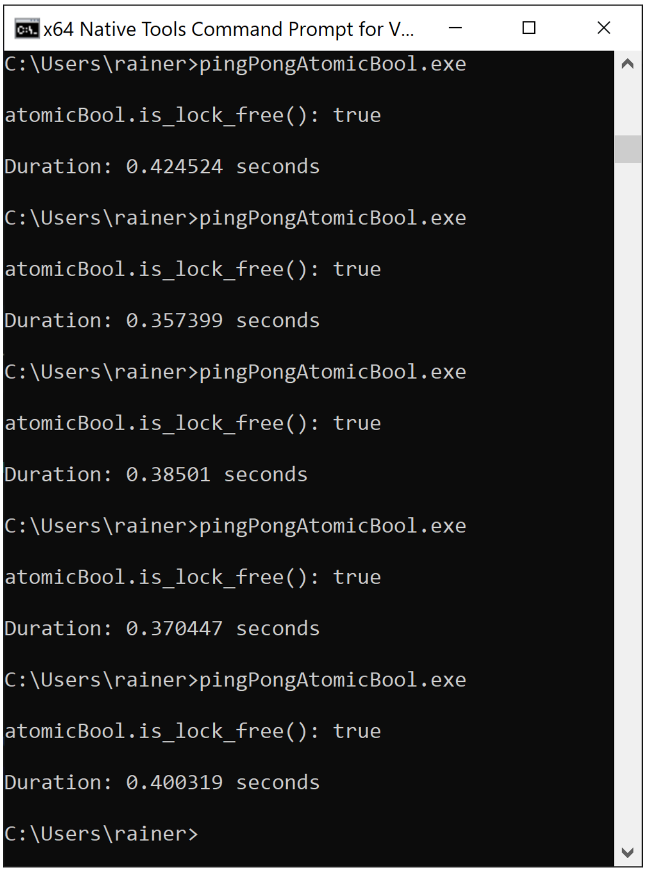
I used in this example `std::atomic_flag` such as an atomic boolean. Let's give it another try with `std::atomic<bool>`.

`std::atomic<bool>`

From the readability perspective, I prefer the following C++20 implementation based on `std::atomic<bool>`.

```cpp
// pingPongAtomicBool.cpp

#include <iostream>
#include <atomic>
#include <thread>

std::atomic<bool> atomicBool{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        atomicBool.wait(true);
        atomicBool.store(true);

        ++counter;

        atomicBool.notify_one();
    }
}

void pong() {
    while(counter < countlimit) {
        atomicBool.wait(false);
        atomicBool.store(false);
        atomicBool.notify_one();
    }
}

int main() {

    std::cout << std::boolalpha << std::endl;

    std::cout << "atomicBool.is_lock_free(): "              // (1)
              << atomicBool.is_lock_free()  << std::endl;

    std::cout << std::endl;

    auto start = std::chrono::system_clock::now();

    atomicBool.store(true);
    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << std::endl;

}
```

`std::atomic<bool>` can internally use a locking mechanism such as a mutex. As I assumed it, my Windows runtime is lock-free (1).



On average, the execution time is 0.38 seconds.

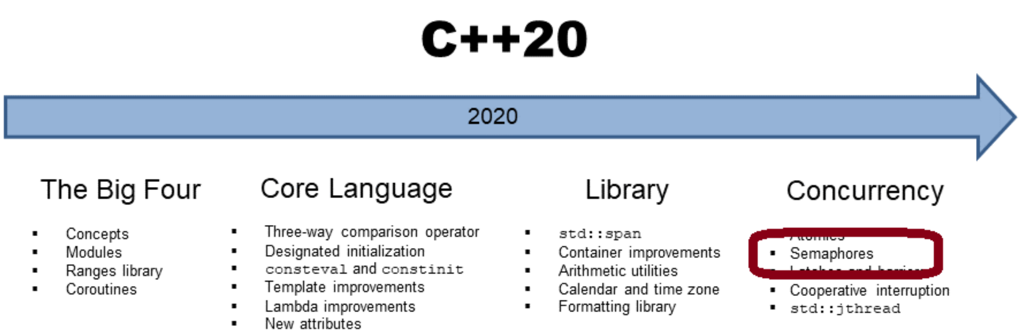
All Numbers

As expected, condition variables are the slowest way, and atomic flag the fastest way to synchronize threads. The performance of a `std::atomic<bool>` is in-between. But there is one downside with `std::.atomic<bool>`. `std::atomic_flag` is the only atomic data type which is lock-free.

Data Type	Condition Variables	Two Atomic Flags	One Atomic Flag	Atomic Bool
Time	0.52	0.32	0.31	0.38

Sempaphores

Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. They also allow it to play ping-pong.



A counting semaphore is a special semaphore that has a counter that is bigger than zero. The counter is initialized in the constructor. Acquiring the semaphore decreases the counter and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Edsger W. Dijkstra Invented Semaphores

The Dutch computer scientist Edsger W. Dijkstra presented in 1965 the concept of a semaphore. A semaphore is a data structure with a queue and a counter. The counter is initialized to a value equal to or greater than zero. It supports the two operations `wait` and `signal`. `wait` acquires the semaphore and decreases the counter; it blocks the thread acquiring the semaphore if the counter is zero. `signal` releases the semaphore and increases the counter. Blocked threads are added to the queue to avoid starvation.

Originally, a semaphore is a railway signal.

The original uploader was AmosWolfe at English Wikipedia. -

Counting Semaphores in C++20

C++20 supports a `std::binary_semaphore`, which is an alias for a `std::counting_semaphore<1>`. In this case, the least maximal value is 1. `std::binary_semaphores` can be used to implement locks.

```
using binary_semaphore = std::counting_semaphore<1>;
```

In contrast to a `std::mutex`, a `std::counting_semaphore` is not bound to a thread. This means, that the acquire and release call of a semaphore can happen on different threads. The following table presents the interface of a `std::counting_semaphore`.

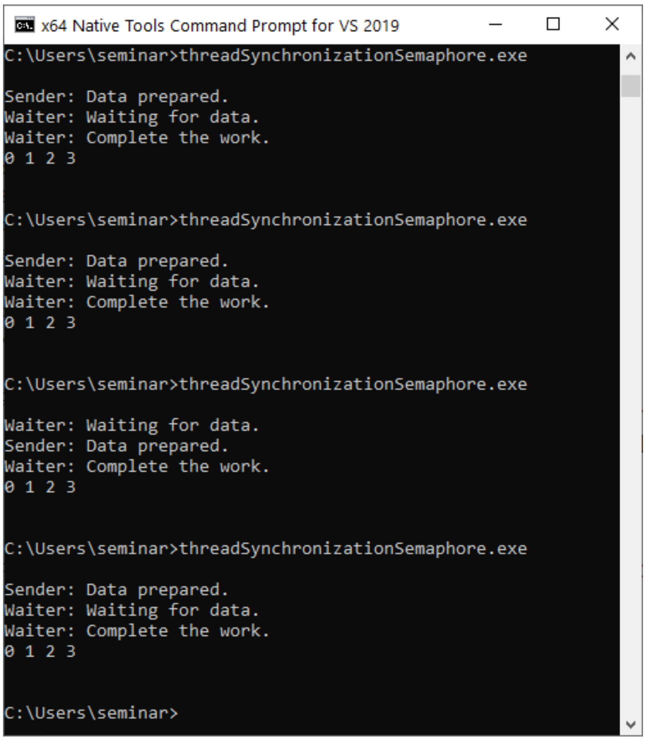
Member function	Description
`sem.max()`	Returns the maximum value of the counter.
`sem.release(upd = 1)`	Increases counter by upd and unblocks subsequently threads acquiring the semaphore sem.
`sem.acquire()`	Decrements counter by 1 or blocks until counter is greater than 0.
`sem.try_acquire()`	Tries to decrement the counter by 1 if it is greater than 0.
`sem.try_acquire_for(relTime)`	Tries to decrement the counter by 1 or blocks for at most relTime if counter is 0.
`sem.try_acquire_until(absTime)`	Tries to decrement the counter by 1 or blocks at most until absTime if counter is 0.

The constructor call `std::counting_semaphore<10> sem(5)` creates a semaphore sem with an at least maximal value of 10 and a counter of 5. The call `sem.max()` returns the least maximal value. `sem.try_aquire_for(relTime)` needs a relative time duration; the member function `sem.try_acquire_until(absTime)` needs an absolute time point. You can read more about time durations and time points in my previous posts to the time libraray: time. The three calls `sem.try_acquire`, `sem.try_acquire_for`, and `sem.try_acquire_until` return a boolean indicating the success of the calls.

Semaphores are typically used in sender-receiver workflows. For example, initializing the semaphore sem with 0 will block the receivers `sem.acquire()` call until the sender calls `sem.release()`. Consequently, the receiver waits for the notification of the sender. A one-time synchronization of threads can easily be implemented using semaphores.

```
// threadSynchronizationSemaphore.cpp

#include <iostream>
#include <semaphore>
#include <thread>
#include <vector>

std::vector<int> myVec{};

std::counting_semaphore<1> prepareSignal(0);              // (1)

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << '\n';
    prepareSignal.release();                              // (2)
}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    prepareSignal.acquire();                              // (3)
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';

}

int main() {

    std::cout << '\n';

    std::thread t1(prepareWork);
    std::thread t2(completeWork);

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

    std::cout << '\n';

}
```

The `std::counting_semaphore prepareSignal` (1) can have the values 0 oder 1. In the concrete example, it's initialized with 0 (line 1). This means, that the call `prepareSignal.release()` sets the value to 1 (line 2) and unblocks the call `prepareSignal.acquire()` (line 3).



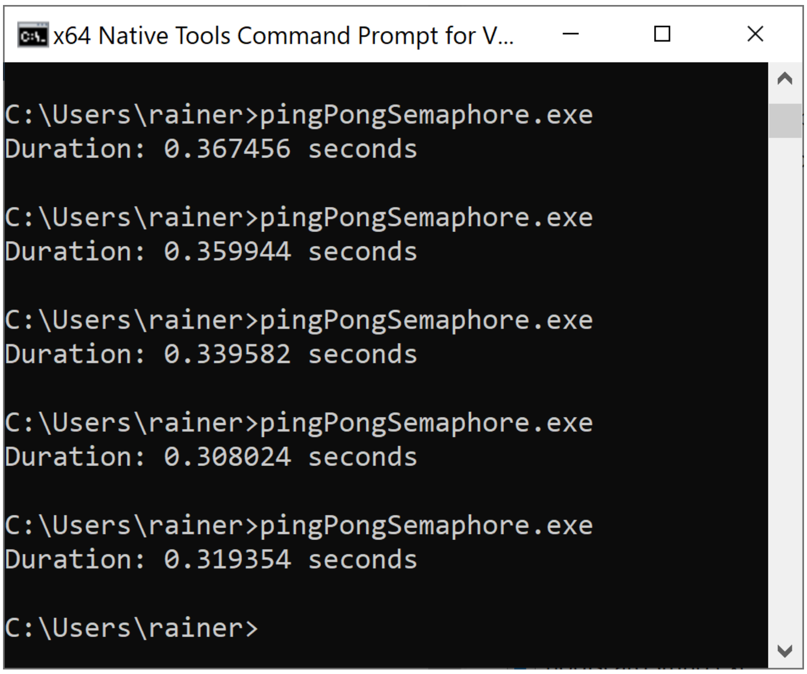
Let me make a small performance test by playing ping-pong with semaphores.

A Ping-Pong Game

In my last post "Performance Comparison of Condition Variables and Atomics in C++20", I implemented a ping-pong game. Here is the idea of the game: One thread executes a `ping` function and the other thread a`pong` function. The ping thread waits for the notification of the pong thread and sends the notification back to the pong thread. The game stops after 1,000,000 ball changes. I perform each game five times to get comparable performance numbers. Let's start the game:

```cpp
// pingPongSemaphore.cpp

#include <iostream>
#include <semaphore>
#include <thread>

std::counting_semaphore<1> signal2Ping(0);       // (1)
std::counting_semaphore<1> signal2Pong(0);       // (2)

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        signal2Ping.acquire();                  // (5)
        ++counter;
        signal2Pong.release();
    }
}

void pong() {
    while(counter < countlimit) {
        signal2Pong.acquire();
        signal2Ping.release();                  // (3)
    }
}

int main() {

    auto start = std::chrono::system_clock::now();

    signal2Ping.release();                     // (4)
    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << '\n';

}
```

The program `pingPongsemaphore.cpp` uses two semaphores: `signal2Ping` and `signal2Pong` (1 and 2). Both can have the two values 0 and 1 and are initialized with 0. This means when the value is 0 for the semaphore `signal2Ping`, a call `signal2Ping.release()` (3 and 4) set the value to 1 and is, therefore, a notification. A `signal2Ping.acquire()` (5) call blocks until the value becomes 1. The same argumentation holds for the second `semaphore signal2Pong`.



On average, the execution time is 0.33 seconds.

Let me summarize the performance numbers for all ping-pong games. This includes the performance numbers of my last post "Performance Comparison of Condition Variables and Atomics in C++20" and this ping-pong game implemented with semaphores.

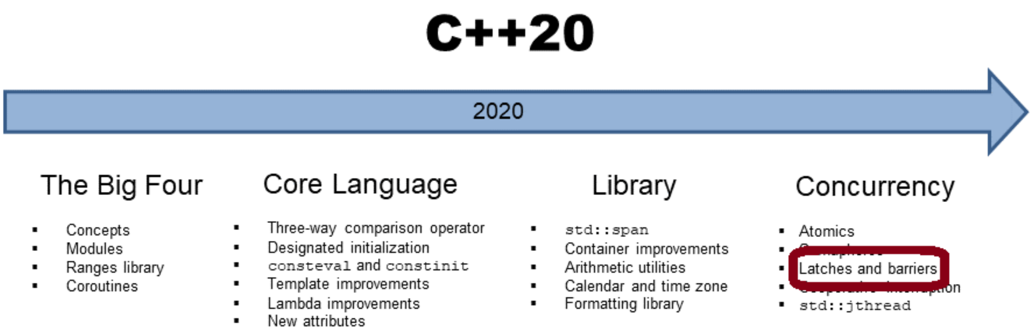
All Numbers

Condition variables are the slowest way, and atomic flag the fastest way to synchronize threads. The performance of a `std::atomic` is in between. There is one downside with `std::atomic`. `std::atomic_flag` is the only atomic data type that is always lock-free. Semaphores impressed me most because they are nearly as fast as atomic flags.

	Condition Variables	Two Atomic Flags	One Atomic Flag	Atomic Boolean	Semaphores
Execution Time	0.52	0.32	0.31	0.38	0.33

Latches

Latches and barriers are coordination types that enable some threads to wait until a counter becomes zero. You can use a `std::latch` only once, but you can use a `std::barrier` more than once. Today, I have a closer look at latches.



Concurrent invocations of the member functions of a `std::latch` or a `std::barrier` are no data race. A data race is such a crucial term in concurrency that I want to write more words to it.

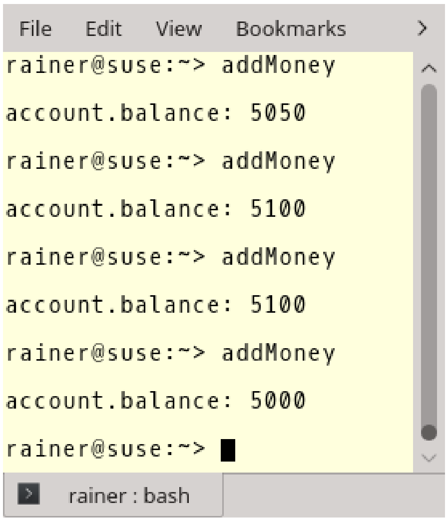
Data Race

A data race is a situation, in which at least two threads access a shared variable at the same time and at least one thread tries to modify the variable. If your program has a data race, it has undefined behavior. This means all outcomes are possible and therefore, reasoning about the program makes no sense anymore.

Let me show you a program with a data race.

```cpp
// addMoney.cpp

#include <functional>
#include <iostream>
#include <thread>
#include <vector>

struct Account{
  int balance{100};                                              // (3)
};

void addMoney(Account& to, int amount){                          // (2)
  to.balance += amount;                                          // (1)
}

int main(){

  std::cout << '\n';

  Account account;

  std::vector<std::thread> vecThreads(100);

  for (auto& thr: vecThreads) thr = std::thread(addMoney, std::ref(account), 50);

  for (auto& thr: vecThreads) thr.join();

  std::cout << "account.balance: " << account.balance << '\n';      // (4)

  std::cout << '\n';

}
```

100 threads adding 50 euros to the same account (1) using the function `addMoney` (2). The initial account is 100 (3). The crucial observation is that the writing to the account is done without synchronization. Therefore we have a data race and, consequently, undefined behavior. The final balance is between 5000 and 5100 euro (4).



What is happening? Why are a few additions missing? The update process `to.balance += amount;` in line (1) is a so-called read-modify-write operation. As such, first, the old value of `to.balance` is read, then it is updated, and finally is written. What may happen under the hood is the following. I use numbers to make my argumentation more obvious

- Thread A reads the value 500 euro and then Thread B kicks in.
- Thread B read also the value 500 euro, adds 50 euro to it, and updates `to.balance` to 550 euro.
- Now Thread A finished its execution by adding 50 euro to `to.balance` and also writes 550 euro.
- Essential the value 550 euro is written twice and instead of two additions of 50 euro, we only observe one.
- This means, that one modification is lost and we get the wrong final sum.

First, there are two questions to answer before I present `std::latch` and `std::barrier` in detail.

Two Questions

1. **What is the difference between these two mechanisms to coordinate threads?** You can use a `std::latch` only once, but you can use a `std::barrier` more than once. A `std::latch` is useful for managing one task by multiple threads; a `std::barrier` is helpful for managing repeated tasks by multiple threads. Additionally, a `std::barrier` enables you to execute a function in the so-called completion step. The completion step is the state when the counter becomes zero.
2. **What use cases do latches and barriers support that cannot be done in C++11 with futures, threads, or condition variables combined with locks?** Latches and barriers address no new use cases, but they are a lot easier to use. They are also more performant because they often use a lock-free mechanism internally.

Let me continue my post with the simpler data type of both.

`std::latch`

Now, let us have a closer look at the interface of a `std::latch`.

Member function	Description
`lat.count_down(upd = 1)`	Atomically decrements the counter by `upd` without blocking the caller.
`lat.try_wait()`	Returns `true` if `counter == 0`.
`lat.wait()`	Returns immediately if `counter == 0`. If not blocks until `counter == 0`.
`lat.arrive_and_wait(upd = 1)`	Equivalent to `count_down(upd); wait();`.

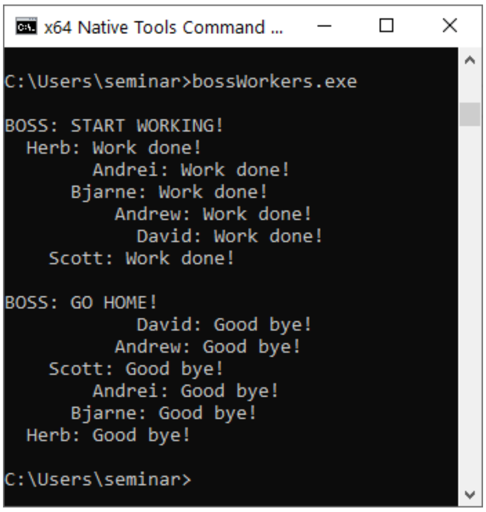
The default value for `upd` is `1`. When `upd` is greater than the counter or negative, the behavior is undefined. The call `lat.try_wait()` does never wait as its name suggests.

The following program `bossWorkers.cpp` uses two `std::latch` to build a boss-workers workflow. I synchronized the output to `std::cout` using the function `synchronizedOut` (1). This synchronization makes it easier to follow the workflow.

```cpp
// bossWorkers.cpp

#include <iostream>
#include <mutex>
#include <latch>
#include <thread>

std::latch workDone(6);
std::latch goHome(1);                                          // (4)

std::mutex coutMutex;

void synchronizedOut(const std::string s) {                    // (1)
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class Worker {
 public:
     Worker(std::string n): name(n) { };

      void operator() (){
          // notify the boss when work is done
          synchronizedOut(name + ": " + "Work done!\n");
          workDone.count_down();                               // (2)

          // waiting before going home
          goHome.wait();                                       // (5)
          synchronizedOut(name + ": " + "Good bye!\n");
      }
 private:
    std::string name;
};

int main() {

    std::cout << '\n';

    std::cout << "BOSS: START WORKING! " << '\n';

    Worker herb("  Herb");
    std::thread herbWork(herb);

    Worker scott("    Scott");
    std::thread scottWork(scott);

    Worker bjarne("      Bjarne");
    std::thread bjarneWork(bjarne);

    Worker andrei("        Andrei");
    std::thread andreiWork(andrei);

    Worker andrew("          Andrew");
    std::thread andrewWork(andrew);

    Worker david("            David");
    std::thread davidWork(david);

    workDone.wait();                                           // (3)

    std::cout << '\n';

    goHome.count_down();

    std::cout << "BOSS: GO HOME!" << '\n';

    herbWork.join();
    scottWork.join();
    bjarneWork.join();
    andreiWork.join();
    andrewWork.join();
    davidWork.join();

}
```

The idea of the workflow is straightforward. The six workers`herb`, `scott`, `bjarne`, `andrei`, `andrew`, and `david` in the `main`-program have to fulfill their job. When they finished their job, they count down the `std::latch workDone` (2). The boss (`main-`

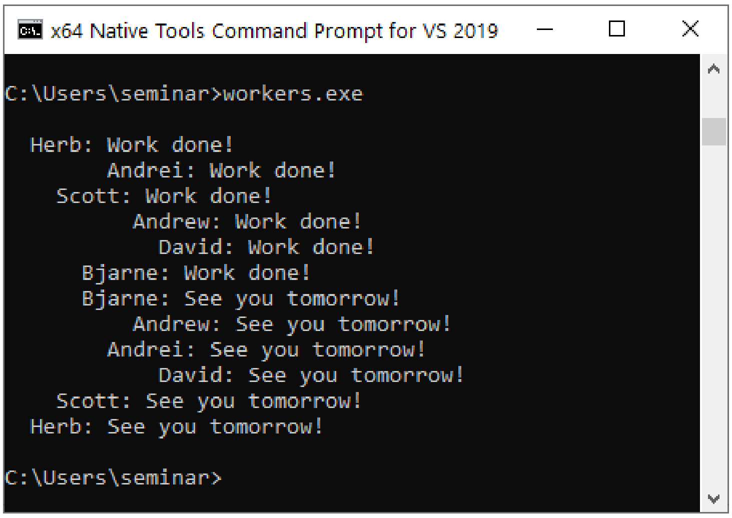
thread) is blocked in line (3) until the counter becomes 0. When the counter is 0, the boss uses the second `std::latch goHome` to signal its workers to go home. In this case, the initial counter is `1` (4). The call `goHome.wait` (5) blocks until the counter becomes 0.



When you think about this workflow, you may notice that it can be performed without a boss. Here is the modern variant:

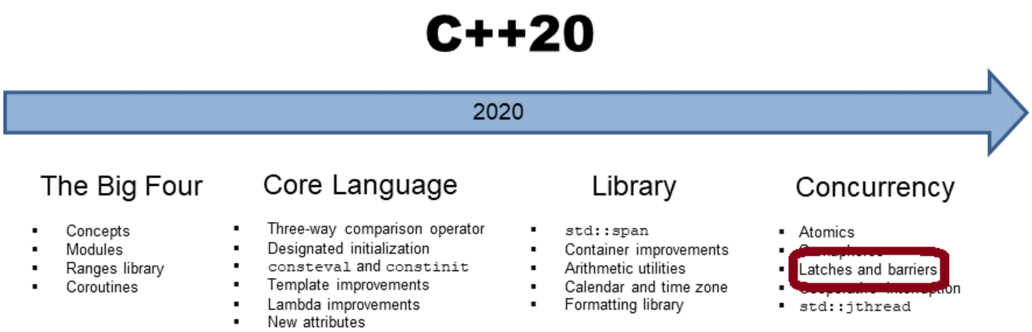
```cpp
// workers.cpp

#include <iostream>
#include <latch>
#include <mutex>
#include <thread>

std::latch workDone(6);
std::mutex coutMutex;

void synchronizedOut(const std::string& s) {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class Worker {
 public:
    Worker(std::string n): name(n) { };

    void operator() () {
        synchronizedOut(name + ": " + "Work done!\n");
        workDone.arrive_and_wait();  // wait until all work is done   (1)
        synchronizedOut(name + ": " + "See you tomorrow!\n");
    }
 private:
    std::string name;
};

int main() {

    std::cout << '\n';

    Worker herb("  Herb");
    std::thread herbWork(herb);

    Worker scott("    Scott");
    std::thread scottWork(scott);

    Worker bjarne("      Bjarne");
    std::thread bjarneWork(bjarne);

    Worker andrei("        Andrei");
    std::thread andreiWork(andrei);

    Worker andrew("          Andrew");
    std::thread andrewWork(andrew);

    Worker david("            David");
    std::thread davidWork(david);

    herbWork.join();
    scottWork.join();
    bjarneWork.join();
    andreiWork.join();
    andrewWork.join();
    davidWork.join();

}
```

There is not much to add to this simplified workflow. The call`workDone.arrive_and_wait(1)` (1) is equivalent to the calls `count_down(upd); wait();`. This means the workers coordinate themself and the boss is no longer necessary such as in the previous program `bossWorkers.cpp`.



Barriers and Atomic Smart Pointers

In my last post, I introduced latches in C++20. A latch enables its threads to wait until a counter becomes zero. Additionally, to a latch, its big sibling barrier can be used more than once. Today, I write about barriers and present atomic smart pointers.



If you are not familiar with std::latch, read my last post:[Latches in C++20](#).

`std::barrier`

There are two differences between a `std::latch` and a `std::barrier`. A `std::latch` is useful for managing one task by multiple threads; a `std::barrier` is helpful for managing repeated tasks by multiple threads. Additionally, a `std::barrier` enables you to execute a function in the so-called completion step. The completion step is the state when the counter becomes zero. Immediately after the counter becomes zero, the so-called completion step starts. In this completion step, a callable is invoked. The `std::barrier` gets its callable in its constructor. A callable unit (short callable) is something that behaves like a function. Not only are these named functions, but also function objects or lambda expressions.

The completion step performs the following steps:

1. All threads are blocked.
2. An arbitrary thread is unblocked and executes the callable.
3. If the completion step is done, all threads are unblocked.

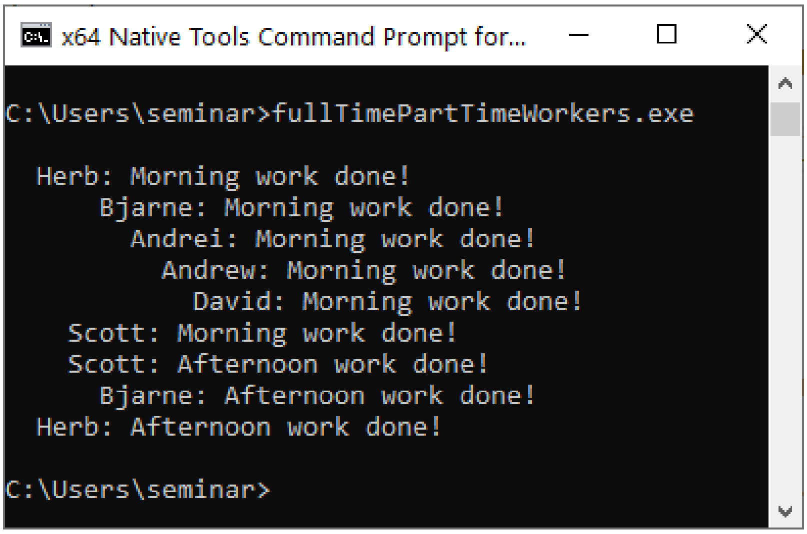
The following table presents you the interface of a `std::barrier bar`.

Member function	Description
`bar.arrive(upd)`	Atomically decrements counter by `upd`.
`bar.wait()`	Blocks at the synchronization point until the completion step is done.
`bar.arrive_and_wait()`	Equivalent to `wait(arrive())`
`bar.arrive_and_drop()`	Decrements the counter for the current and the subsequent phase by one.
`std::barrier::max`	Maximum value supported by the implementation.

The `call bar.arrive_and_drop()` call means essentially, that the counter is decremented by one for the next phase. The following program `fullTimePartTimeWorkers.cpp` halves the number of workers in the second phase.

```cpp
// fullTimePartTimeWorkers.cpp

#include <iostream>
#include <barrier>
#include <mutex>
#include <string>
#include <thread>

std::barrier workDone(6);
std::mutex coutMutex;

void synchronizedOut(const std::string& s) noexcept {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class FullTimeWorker {                                          // (1)
 public:
    FullTimeWorker(std::string n): name(n) { };

    void operator() () {
        synchronizedOut(name + ": " + "Morning work done!\n");
        workDone.arrive_and_wait();  // Wait until morning work is done     (3)
        synchronizedOut(name + ": " + "Afternoon work done!\n");
        workDone.arrive_and_wait();  // Wait until afternoon work is done   (4)

    }
 private:
    std::string name;
};

class PartTimeWorker {                                          // (2)
 public:
    PartTimeWorker(std::string n): name(n) { };

    void operator() () {
        synchronizedOut(name + ": " + "Morning work done!\n");
        workDone.arrive_and_drop();  // Wait until morning work is done  // (5)
    }
 private:
    std::string name;
};

int main() {

    std::cout << '\n';

    FullTimeWorker herb("  Herb");
    std::thread herbWork(herb);

    FullTimeWorker scott("    Scott");
    std::thread scottWork(scott);

    FullTimeWorker bjarne("      Bjarne");
    std::thread bjarneWork(bjarne);

    PartTimeWorker andrei("        Andrei");
    std::thread andreiWork(andrei);

    PartTimeWorker andrew("          Andrew");
    std::thread andrewWork(andrew);

    PartTimeWorker david("            David");
    std::thread davidWork(david);

    herbWork.join();
    scottWork.join();
    bjarneWork.join();
    andreiWork.join();
    andrewWork.join();
    davidWork.join();

}
```

This workflow consists of two kinds of workers: full-time workers (1) and part-time workers (2). The part-time worker works in the morning, the full-time worker in the morning and the afternoon. Consequently, the full-time workers call `workDone.arrive_and_wait()` (lines (3) and (4)) two times. On the contrary, the part-time works call `workDone.arrive_and_drop()` (5) only once. This `workDone.arrive_and_drop()` call causes the part-time worker to skip the afternoon work. Accordingly, the counter has in the first phase (morning) the value 6, and in the second phase (afternoon) the value 3.



Now to something, I missed in my posts to atomics.

Atomic Smart Pointers

A `std::shared_ptr` consists of a control block and its resource. The control block is thread-safe, but access to the resource is not. This means modifying the reference counter is an atomic operation and you have the guarantee that the resource is deleted exactly once. These are the guarantees `std::shared_ptr` gives you.

On the contrary, it is crucial that a `std::shared_ptr` has well-defined multithreading semantics. At first glance, the use of a `std::shared_ptr` does not appear to be a sensible choice for multithreaded code. It is by definition shared and mutable and is the ideal candidate for non-synchronized read and write operations and hence for undefined behaviour. On the other hand, there is the guideline in modern C++: **Don't use raw pointers**. This means, consequently, that you should use smart pointers in multithreading programs when you want to model shared ownership.

The proposal [N4162](N4162) for atomic smart pointers directly addresses the deficiencies of the current implementation. The deficiencies boil down to these three points: consistency, correctness, and performance.

- **Consistency**: the atomic operations `std::shared_ptr` are the only atomic operations for a non-atomic data type.
- **Correctness**: the usage of the global atomic operations is quite error-prone because the correct usage is based on discipline. It is easy to forget to use an atomic operation - such as using `ptr = localPtr` instead of `std::atomic_store(&ptr, localPtr)`. The result is undefined behaviour because of a data race. If we used an atomic smart pointer instead, the type system would not allow it.
- **Performance**: the atomic smart pointers have a big advantage compared to the free `atomic_*` functions. The atomic versions are designed for the special use case and can internally have a `std::atomic_flag` as a kind of cheap spinlock. Designing the non-atomic versions of the pointer functions to be thread-safe would be overkill if they are used in a single-threaded scenario. They would have a performance penalty.

The correctness argument is probably the most important one. Why? The answer lies in the proposal. The proposal presents a thread-safe singly linked list that supports insertion, deletion, and searching of elements. This singly linked list is implemented in a lock-free way.

```
template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
          // in C++11: remove “atomic_” and remember to use the special
          // functions every time you touch the variable
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
       reference(shared_ptr<Node> p_) : p{p_} { }
       T& operator* () { return p->t; }
       T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load();  // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
       return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;         // in C++11: atomic_load(&head)
      while( !head.compare_exchange_weak(p->next, p) ){ }
      // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
    }
    void pop_front() {
       auto p = head.load();
       while( p && !head.compare_exchange_weak(p, p->next) ){ }
       // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};
```

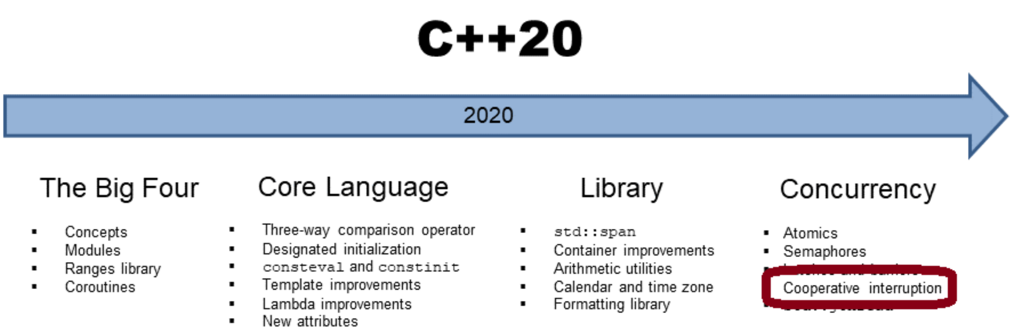
All changes that are required to compile the program with a C++11 compiler are marked in red. The implementation with atomic smart pointers is a lot easier and hence less error-prone. C++20's type system does not permit it to use a non-atomic operation on an atomic smart pointer.

The proposal N4162 proposed the new types `std::atomic_shared_ptr` and `std::atomic_weak_ptr` as atomic smart pointers. By merging them in the mainline ISO C++ standard, they became partial template specialization of std::atomic: `std::atomic<std::shared_ptr>`, and `std::atomic<std::weak_ptr>`.

Consequently, the atomic operations for `std::shared_ptr<T>` are deprecated with C++20.

Cooperative Interruption of a Thread

A typical question in my C++ seminars is: Can A thread be killed?. Before C++20, my answer is no. With C++20, you can ask a thread politely for its interruption.



First of all. Why is it no good idea to kill a thread? The answer is quite easy. You don't know in which state the thread is when you kill it. Here are two possible malicious outcomes.

- The thread is only half-done with its job. Consequently, you don't know the state of that job and, hence, the state of your program. You end with undefined behavior, and all bets are open.
- The thread may be in a critical section and locks a mutex. Killing a thread while it locks a mutex ends with a high probability in a deadlock.

Okay, killing a thread is not a good idea. Maybe, you can ask a thread friendly if it is willing to stop. This is exactly what cooperative interruption in C++20 means. You ask the thread, and the thread can accept or ignore your wish for the interruption.

Cooperative Interruption

The additional functionality of the cooperative interruption thread in C++20 is based on the`std::stop_token`, the `std::stop_callback`, and the `std::stop_source` data types.

`std::stop_token`, `std::stop_callback`, and `std::stop_source`

A `std::stop_token`, a `std::stop_callback`, or a `std::stop_source` allows a thread to asynchronously request an execution to stop or ask if an execution got a stop signal. The `std::stop_token` can be passed to an operation and afterward be used to poll the token for a stop request actively or to register a callback via `std::stop_callback`. The stop request is sent by a `std::stop_source`. This signal affects all associated `std::stop_token`. The three classes `std::stop_source`, `std::stop_token`, and `std::stop_callback` share the ownership of an associated stop state. The calls`request_stop()`, `stop_requested()`, and `stop_possible()` are atomic.

You can construct a `std::stop_source` in two ways:

```
stop_source();                                      // (1)
explicit stop_source(std::nostopstate_t) noexcept;  // (2)
```

The default constructor (1) constructs a `std::stop_source` with a new stop state. The constructor taking `std::nostopstate_t` (2) constructs an empty `std::stop_source` without associated stop state.
The component `std::stop_source src` provides the following member functions for handling stop requests.

Member function	Description
`src.get_token()`	If `stop_possible()`, returns a `stop_token` for the associated stop state. Otherwise, returns a default-constructed (empty) `stop_token`.
`src.stop_possible()`	`true` if `src` can be requested to stop.
`src.stop_requested()`	`true` if `stop_possible()` and `request_stop()` was called by one of the owners.
`src.request_stop()`	Calls a stop request if `stop_possible()` and `!stop_requested()`. Otherwise, the call has no effect.

`src.stop_possible()` means that `src` has an associated stop state. `src.stop_requested()` returns `true` when `src` has an associated stop state and was not asked to stop earlier. `src.request_stop()` is successful and returns `true` if `src` has an associated stop state, and it was not requested to stop before.

The call `src.get_token()` returns the stop token `stoken`. Thanks to `stoken` you can check if a stop request has been made or can be made for its associated stop source `src`. The stop token `stoken` observes the stop source `src`.

The following table presents the member functions of a`std::stop_token stoken`.

Member function	Description
`stoken.stop_possible()`	Returns `true` if `stoken` has an associated stop state.
`stoken.stop_requested()`	`true` if `request_stop()` was called on the associated `std::stop_source src`, otherwise `false`.

A default-constructed token that has no associated stop state. `stoken.stop_possible` also returns `true` if `stoken` has an associated stop state. `stoken.stop_requested()` returns `true` when stop token has an associated stop state and has already received a stop request.

If the `std::stop_token` should be temporarily disabled, you can replace it with a default constructed token. A default constructed token has no associated stop-state. The following code snippet shows how to disable and enable a thread's capability to accept stop requests.

```
std::jthread jthr([](std::stop_token stoken) {
    ...
    std::stop_token interruptDisabled;
    std::swap(stoken, interruptDisabled);  // (1)
    ...                                    // (2)
    std::swap(stoken, interruptDisabled);
    ...
}
```

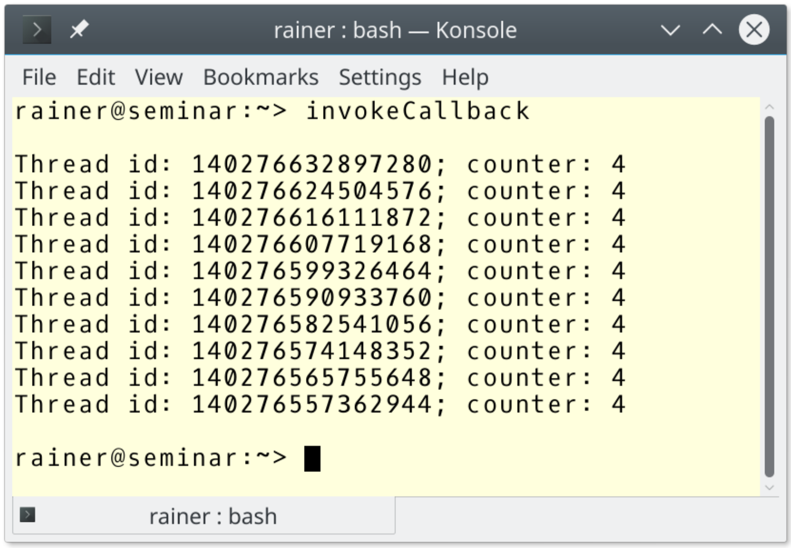
`std::stop_token interruptDisabled` has no associated stop state. This means the thread `jthr` can in all lines except line (1) and (2) accept stop requests.

When you study the code snippet carefully, you may wonder about the used`std::jthread`. `std::jthread` in C++20 is an extend `std::thread` in C++11. The **j** in `jthread` stands for joinable because it joins automatically in its destructor. Its first name was `ithread`. You may guess why: **i** stands for interruptable. I present`std::jthread` in my next post.

My next example shows the use of callbacks using a`std::jthread`.

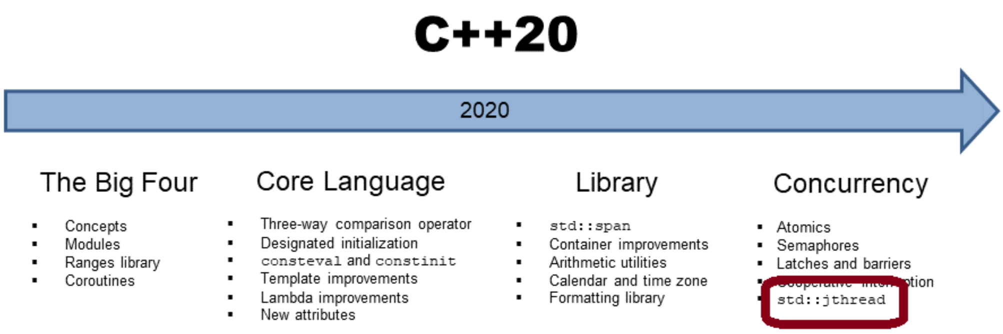
```
// invokeCallback.cpp

#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

using namespace::std::literals;

auto func = [](std::stop_token stoken) {                          // (1)
        int counter{0};
        auto thread_id = std::this_thread::get_id();
        std::stop_callback callBack(stoken, [&counter, thread_id] {  // (2)
            std::cout << "Thread id: " << thread_id
                      << "; counter: " << counter << '\n';
        });
        while (counter < 10) {
            std::this_thread::sleep_for(0.2s);
            ++counter;
        }
    };

int main() {

    std::cout << '\n';

    std::vector<std::jthread> vecThreads(10);
    for(auto& thr: vecThreads) thr = std::jthread(func);

    std::this_thread::sleep_for(1s);                           // (3)

    for(auto& thr: vecThreads) thr.request_stop();             // (4)

    std::cout << '\n';

}
```

Each of the ten threads invokes the lambda function `func` (1). The callback (2) displays the thread `id` and the `counter`. Due to the one-second sleeping of the main-thread (3) and the sleeping of the child threads, the counter is 4 when the callbacks are invoked. The call `thr.request_stop()` triggers the callback on each thread.



An improved Thread

`std::jthread` stands for joining thread. In addition to `std::thread` (C++11), `std::jthread` automatically joins in its destructor and can cooperatively be interrupted. Read in this post to know, why `std::jthread` should be your first choice.



The following table gives you a concise overview of the functionality of `std::jthread`.

Functions	Description
`t.join()`	Waits until thread `t` has finished its execution.
`t.detach()`	Executes the created thread `t` independently of the creator.
`t.joinable()`	Returns true if thread `t` is still joinable.
`t.get_id()` and `std::this_thread::get_id()`	Returns the id of the thread.
`std::jthread::hardware_concurrency()`	Indicates the number of threads that can run concurrently.
`std::this_thread::sleep_until(absTime)`	Puts thread `t` to sleep until the time point `absTime`.
`std::this_thread::sleep_for(relTime)`	Puts thread `t` to sleep for the time duration `relTime`.
`std::this_thread::yield()`	Enables the system to run another thread.
`t.swap(t2)` and `std::swap(t1, t2)`	Swaps the threads.
`t.get_stop_source()`	Returns a `std::stop_source` object associated with the shared stop state.
`t.get_stop_token()`	Returns a `std::stop_token` object associated with the shared stop state.
`t.request_stop()`	Requests execution stop via the shared stop state.

For additional details, please refer to cppreference.com. When you want to read more post about `std::thread`, here are they: my post about std::thread.

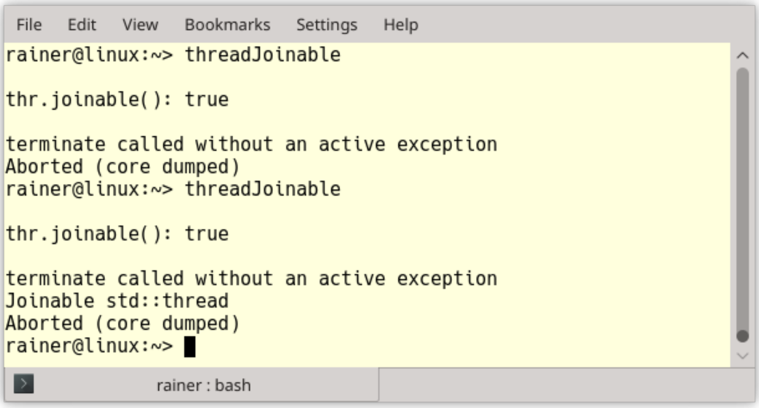
First, why do we need an improved thread in C++20? Here is the first reason.

Automatically Joining

This is the **non-intuitive** behavior of `std::thread`. If a `std::thread` is still joinable, std::terminate is called in its destructor. A thread `thr` is joinable if neither `thr.join()` nor `thr.detach()` was called. Let me show, what that means.

```
// threadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

    std::cout << '\n';
    std::cout << std::boolalpha;

    std::thread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```

When executed, the program terminates when the local object `thr` goes out of scope.

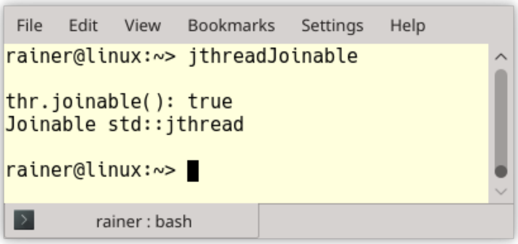


Both executions of `std::thread` terminate. In the second run, the thread `thr` has enough time to display its message: `Joinable std::thread`.

In the next example, I use `std::jthread` from the C++20 standard.

```
// jthreadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

    std::cout << '\n';
    std::cout << std::boolalpha;

    std::jthread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```

Now, the thread `thr` automatically joins in its destructor if it's still joinable such as in this case.



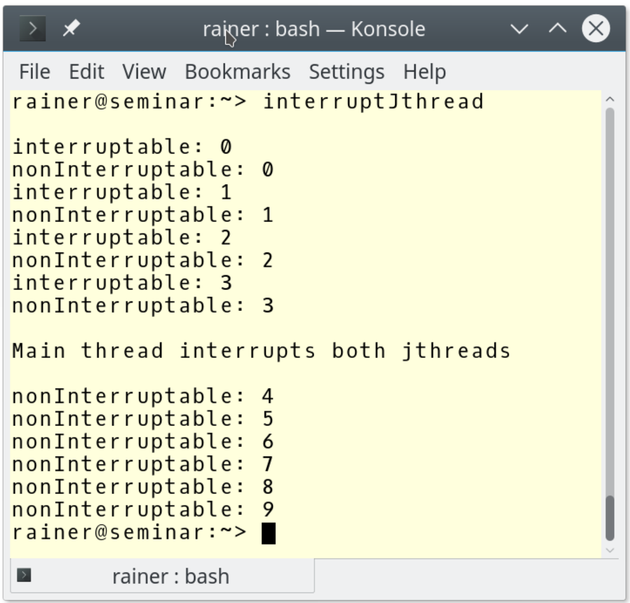
But this is not all that `std::jthread` provides additionally to `std::thread`. A `std::jthread` can be cooperatively interrupted. I already presented the general ideas of cooperative interruption in my last post: [Cooperative Interruption of a Thread in C++20](#).

Cooperative Interruption of a `std::jthread`

To get a general idea, let me present a simple example.

```cpp
// interruptJthread.cpp

#include <chrono>
#include <iostream>
#include <thread>

using namespace::std::literals;

int main() {

    std::cout << '\n';

    std::jthread nonInterruptable([]{                           // (1)
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);
            std::cerr << "nonInterruptable: " << counter << '\n';
            ++counter;
        }
    });

    std::jthread interruptable([](std::stop_token stoken){      // (2)
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);
            if (stoken.stop_requested()) return;                // (3)
            std::cerr << "interruptable: " << counter << '\n';
            ++counter;
        }
    });

    std::this_thread::sleep_for(1s);

    std::cerr << '\n';
    std::cerr << "Main thread interrupts both jthreads" << '\n';
    nonInterruptable.request_stop();
    interruptable.request_stop();                               // (4)

    std::cout << '\n';

}
```

In the main program, I start the two threads`nonInterruptable` and interruptable (lines 1)and 2). Unlike in the thread `nonInterruptable` , the thread `interruptable` gets a `std::stop_token` and uses it in line (3) to check if it was interrupted: `stoken.stop_requested()`. In case of a stop request, the lambda function returns, and, therefore, the thread ends. The call `interruptable.request_stop()` (line 4) triggers the stop request. This does not hold for the previous call `nonInterruptable.request_stop()` . The call has no effect.



To make my post complete, with C++20, you can also cooperatively interrupt a condition variable.

New wait Overloads for `std::condition_variable_any`

Before I write about std::condition_variable_any, here are my post about condition variables.

The three wait variations wait, wait_for, and wait_until of the std::condition_variable_any get new overloads. These overloads take a std::stop_token.

```
template <class Predicate>
bool wait(Lock& lock,
          stop_token stoken,
          Predicate pred);

template <class Rep, class Period, class Predicate>
bool wait_for(Lock& lock,
              stop_token stoken,
              const chrono::duration<Rep, Period>& rel_time,
              Predicate pred);

template <class Clock, class Duration, class Predicate>
bool wait_until(Lock& lock,
                stop_token stoken,
                const chrono::time_point<Clock, Duration>& abs_time,
                Predicate pred);
```

These new overloads need a predicate. The presented versions ensure to get notified if a stop request for the passed std::stop_token stoken is signaled. They return a boolean that indicates whether the predicate evaluates to true. This returned boolean is independent of whether a stop was requested or of whether the timeout was triggered.

After the wait calls, you can check if a stop request occurred.

```
cv.wait(lock, stoken, predicate);
if (stoken.stop_requested()){
    // interrupt occurred
}
```

The following example shows the usage of a condition variable with a stop request.

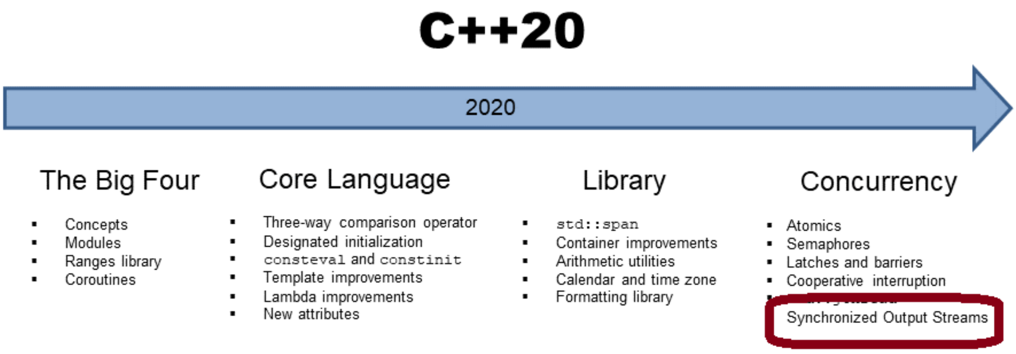
```cpp
// conditionVariableAny.cpp

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

using namespace std::literals;

std::mutex mutex_;
std::condition_variable_any condVar;

bool dataReady;

void receiver(std::stop_token stopToken) {                    // (1)

    std::cout << "Waiting" << '\n';

    std::unique_lock<std::mutex> lck(mutex_);
    bool ret = condVar.wait(lck, stopToken, []{return dataReady;});
    if (ret){
        std::cout << "Notification received: " << '\n';
    }
    else{
         std::cout << "Stop request received" << '\n';
    }
}

void sender() {                                               // (2)

    std::this_thread::sleep_for(5ms);
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
        std::cout << "Send notification"  << '\n';
    }
    condVar.notify_one();                                     // (3)

}

int main(){

  std::cout << '\n';

  std::jthread t1(receiver);
  std::jthread t2(sender);

  t1.request_stop();                                          // (4)

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

  std::cout << '\n';

}
```

The receiver thread (line 1) is waiting for the notification of the sender thread (line 2). Before the sender thread sends its notification (line 3), the main thread triggered a stop request in
line (4). The output of the program shows that the stop request happened before the notification.

```
Waiting
Stop request received
Send notification
```

Synchronized Output Streams

What happens when you write without synchronization to `std::cout`? You get a mess. With C++20, this must not be anymore.

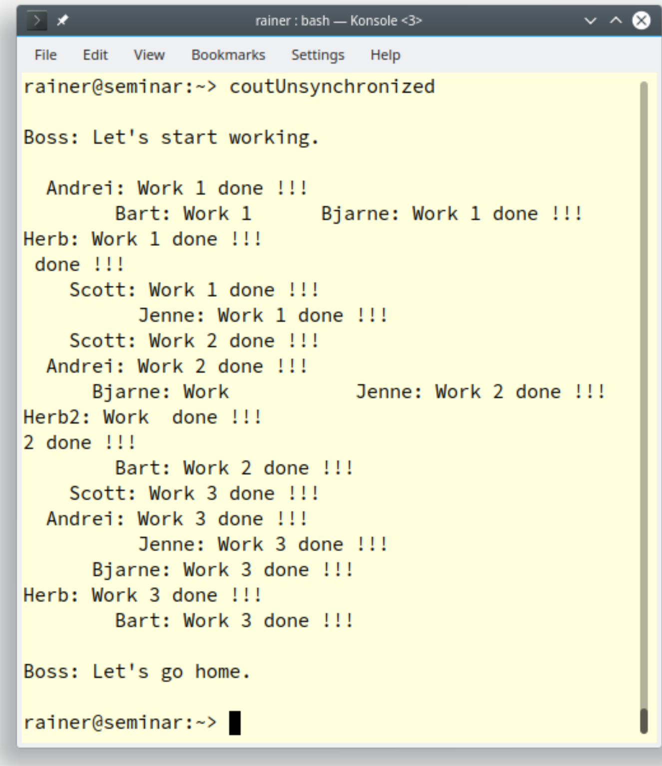


Before I present synchronized output streams with C++20, I want to show non-synchronized output in C++11.

```
// coutUnsynchronized.cpp

#include <chrono>
#include <iostream>
#include <thread>

class Worker{
public:
  Worker(std::string n):name(n) {};
    void operator() (){
      for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));      // (3)
        // end work
        std::cout << name << ": " << "Work " << i << " done !!!" << '\n'; // (4)
      }
    }
private:
  std::string name;
};


int main() {

  std::cout << '\n';

  std::cout << "Boss: Let's start working.\n\n";

  std::thread herb= std::thread(Worker("Herb"));                         // (1)
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread bart= std::thread(Worker("        Bart"));
  std::thread jenne= std::thread(Worker("          Jenne"));             // (2)


  herb.join();
  andrei.join();
  scott.join();
  bjarne.join();
  bart.join();
  jenne.join();

  std::cout << "\n" << "Boss: Let's go home." << '\n';                   // (5)

  std::cout << '\n';

}
```

The boss has six workers (lines 1 - 2). Each worker has to take care of three work packages that take 1/5 second each (line 3). After the worker is done with his work package, he screams out loudly to the boss (line 4). Once the boss receives

notifications from all workers, he sends them home (line 5).
What a mess for such a simple workflow! Each worker screams out his message ignoring his coworkers!

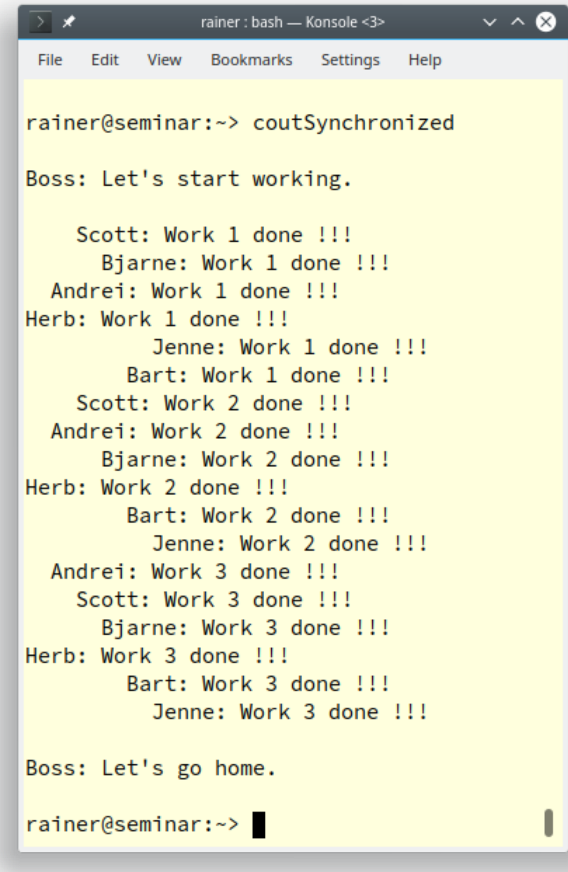


- **`std::cout` is thread-safe**: The C++11 standard guarantees that you must not protect`std::cout`. Each character is written atomically. More output statements like those in the example may interleave. This interleaving is only a visual issue; the program is well-defined. This remark is valid for all global stream objects. Insertion to and extraction from global stream objects (`std::cout, std::cin, std::cerr`, and `std::clog`) is thread-safe. To put it more formally: writing to `std::cout` is not participating in a data race, but does create a race condition. This means that the output depends on the interleaving of threads. Read more about the terms data race and race condition in my previous post: Race Conditions versus Data Races.

How can we solve this issue? With C++11, the answer is straightforward: use a lock such as`std::lock_guard` to synchronize the access to `std::cout`. For more information about locks in C++11, read my previous postPrefer Locks to Mutexes.

```cpp
// coutSynchronized.cpp

#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex coutMutex;                                                          // (1)

class Worker{
public:
  Worker(std::string n):name(n) {};

    void operator() (){
      for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::lock_guard<std::mutex> coutLock(coutMutex);                     // (2)
        std::cout << name << ": " << "Work " << i << " done !!!" << '\n';
      }                                                                        // (3)
    }
private:
  std::string name;
};


int main() {

  std::cout << '\n';

  std::cout << "Boss: Let's start working." << "\n\n";

  std::thread herb= std::thread(Worker("Herb"));
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread bart= std::thread(Worker("        Bart"));
  std::thread jenne= std::thread(Worker("          Jenne"));

  herb.join();
  andrei.join();
  scott.join();
  bjarne.join();
  bart.join();
  jenne.join();

  std::cout << "\n" << "Boss: Let's go home." << '\n';

  std::cout << '\n';

}
```

The `coutMutex` in line (1) protects the shared object `std::cout`. Putting the `coutMutex` into a `std::lock_guard` guarantees that the `coutMutex` is locked in the constructor (line 2) and unlocked in the destructor (line 3) of the `std::lock_guard`. Thanks to the `coutMutex` guarded by the `coutLock` the mess becomes a harmony.



With C++20, writing synchronized to `std::cout` is a piece of cake. `std::basic_sync`buf is a wrapper for a `std::basic_streambuf`. It accumulates output in its buffer. The wrapper sets its content to the wrapped buffer when it is destructed. Consequently, the content appears as a contiguous sequence of characters, and no interleaving of characters can happen.

Thanks to `std::basic_osyncstream`, you can directly write synchronously to `std::cout` by using a named synchronized output stream.

Here is how the previous program `coutUnsynchronized.cpp` is refactored to write synchronized to `std::cout`. So far, only GCC 11 supports synchronized output streams.

```cpp
// synchronizedOutput.cpp

#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

class Worker{
public:
  Worker(std::string n): name(n) {};
    void operator() (){
      for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::osyncstream syncStream(std::cout);                    // (1)
        syncStream << name << ": " << "Work " << i                 // (3)
                   << " done !!!" << '\n';
      }                                                            // (2)
    }
private:
  std::string name;
};


int main() {

  std::cout << '\n';

  std::cout << "Boss: Let's start working.\n\n";

  std::thread herb= std::thread(Worker("Herb"));
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread bart= std::thread(Worker("        Bart"));
  std::thread jenne= std::thread(Worker("          Jenne"));


  herb.join();
  andrei.join();
  scott.join();
  bjarne.join();
  bart.join();
  jenne.join();

  std::cout << "\n" << "Boss: Let's go home." << '\n';

  std::cout << '\n';

}
```

The only change to the previous program `coutUnsynchronized.cpp` is that `std::cout` is wrapped in a `std::osyncstream` (line 1). When the `std::osyncstream` goes out of scope in line (2), the characters are transferred and `std::cout` is flushed. It is worth mentioning that the `std::cout` calls in the main program do not introduce a data race and, therefore, need not be synchronized. The output happens before or after the output of the threads.

Because I use the `syncStream` declared on line (3) only once, a temporary object may be more appropriate. The following code snippet presents the modified call operator:

```cpp
void operator()() {
  for (int i = 1; i <= 3; ++i) {
    // begin work
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // end work
    std::osyncstream(std::cout) << name << ": " << "Work " << i << " done !!!"
                                << '\n';
  }
}
```

`std::basic_osyncstream syncStream` offers two interesting member functions.

- `syncStream.emit()` emits all buffered output and executes all pending flushes.
- `syncStream.get_wrapped()` returns a pointer to the wrapped buffer.

cppreference.com shows how you can sequence the output of different output streams with the `get_wrapped` member function.

```
// sequenceOutput.cpp

#include <syncstream>
#include <iostream>
int main() {

  std::osyncstream bout1(std::cout);
  bout1 << "Hello, ";
  {
      std::osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
  } // emits the contents of the temporary buffer

  bout1 << "World!" << '\n';

} // emits the contents of bout1
```

```
Goodbye, Planet!
Hello, World!
```