



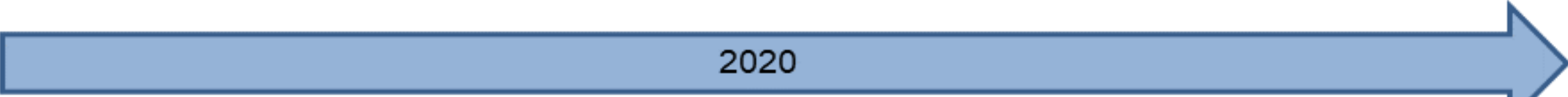
# Concurrency Improvements in C++20: A Deep Dive

Rainer Grimm

Training, Mentoring, and  
Technology Consulting

# C++20 - Concurrency

2020



## The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

## Core Language

- Three-way comparison operator
- Designated initialization
- `constexpr` and `constexpr`
- Template improvements
- Lambda improvements
- New attributes

## Library

- `std::span`
- Container improvements
- Arithmetic utilities
- Calendar and time zone
- Formatting library

## Concurrency

- Atomics
- Semaphores
- Latches and barriers
- Cooperative interruption
- `std::jthread`

# Atomics

Atomics are the foundation of the C++ memory model

➡ Operations on atomics define the synchronization and ordering constraints

- Synchronization and ordering constraints hold for atomics and non-atomics
- Synchronization and ordering constraints are used by the high-level threading interface
  - Threads and tasks
  - Mutexe and locks
  - Condition variables
  - ...

# Atomics

- The atomic flag `std::atomic_flag`
  - Has a very simple interface (`clear` and `test_and_set`).
  - Is the only data type guaranteed to be lock free.

- `std::atomic`

`std::atomic<T*>`

`std::atomic<integral types>`

`std::atomic<user-defined types>`

`std::atomic<floating points>` (C++20)

`std::atomic<smart pointers>` (C++20)

# Atomics

Operation ( <code>std::atomic_flag</code> )	Description
<code>test_and_set</code>	Sets the value and returns the previous value.
<code>clear</code>	Clears the value.

Operation ( <code>std::atomic</code> )	Description
<code>is_lock_free</code>	Checks if the atomic object is lock-free.
<code>load</code>	Returns the value of the atomic.
<code>store</code>	Replaces the value of the atomic with the non-atomic.
<code>exchange</code>	Replaces the value with the new value. Returns the old value.
<code>compare_exchange_weak</code> <code>compare_exchange_strong</code>	<code>atom.compare_exchange_strong(expect, desir)</code> <ul style="list-style-type: none"><li>▪ If <code>atom</code> is equal to <code>expect</code> returns <code>true</code>, <code>atom</code> becomes <code>desir</code>.</li><li>▪ If not returns <code>false</code>, <code>expect</code> is updated with <code>atom</code>.</li></ul>
<code>fetch_add, +=</code> <code>fetch_sub, -=</code>	Adds (subtracts) the value and returns the previous value.
<code>++, --</code>	Increments or decrements the atomic.

[fetch\\_mult.cpp](#)

# Atomics (C++20)

- `std::atomic_flag` and `std::atomic`
  - Enable synchronization of threads
    - `atom.notify_one()`: Notifies one waiting operation
    - `atom.notify_all()`: Notifies all waiting operations
    - `atom.wait(val)`: Waits for a notification and blocks if `atom == val`
  - The default constructor initializes the value.

# Atomics (C++20)

C++11 has `std::shared_ptr` for shared ownership.

- General rule: use smart pointers
- But:
  - The handling of the control block is thread-safe.
  - Access to the resource is not thread-safe.
- Solution in C++20:
  - `std::atomic<std::shared_ptr>`
  - `std::atomic<std::weak_ptr>`

# Atomics

Three reasons for atomic smart pointers.

- Consistency
  - `std::shared_ptr` is the only non-atomic type that supports atomic operations
- Correctness
  - The correct use of the atomic operation weighs on the shoulder of the user
    - ➔ very error-prone
  - ```
std::atomic_store(&sharPtr, localPtr) != sharPtr = localPtr
```
- Speed
  - `std::shared_ptr` is designed for the general use



# Atomics (C++20)

`std::atomic_ref` (C++20) applies atomic operations to the referenced object

- Writing and reading of the referenced object is no data race
  - The lifetime of the referenced object must exceed the lifetime of `std::atomic_ref`
  - `std::atomic_ref` provides the same interface as `std::atomic`
- 
- `std::atomic_ref`
    - `std::atomic_ref<T*>`
    - `std::atomic_ref<integral types>`
    - `std::atomic_ref<user-defined types>`
    - `std::atomic_ref<floating points>`

[atomicReference.cpp](#) ([sanitize](#))

# Semaphores (C++20)

Semaphores are synchronization mechanisms to control access to a shared variable.

A semaphore is initialized with a counter greater than 0

- Requesting the semaphore decrements the counter
  - Releasing the semaphores increments the counter
  - A requesting thread is blocked if the counter is 0
- 
- C++20 support two semaphores.
    - `std::counting_semaphore`
    - `std::binary_semaphore (std::counting_semaphore<1>)`

# Semaphores (C++20)

| Member Function                             | Description                                                                                                                          |
|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <code>counting_semaphore::max()</code>      | Returns the maximum value of the <code>counter</code> .                                                                              |
| <code>sem.release(upd = 1)</code>           | Increases the counter by <code>upd</code> and unblocks threads acquiring the semaphore.                                              |
| <code>sem.acquire()</code>                  | Decrements counter by 1. Blocks if the <code>counter</code> is 0.                                                                    |
| <code>sem.try_acquire()</code>              | Tries to decrement the <code>counter</code> by 1. Don't block if the <code>counter</code> is 0.                                      |
| <code>sem.try_acquire_for(relTime)</code>   | Decrement the <code>counter</code> by 1. Blocks at most for the time duration <code>relTime</code> if the <code>counter</code> is 0. |
| <code>sem.try_acquire_until(absTime)</code> | Decrement the <code>counter</code> by 1. Blocks at most until the time point <code>absTime</code> if <code>counter</code> is 0.      |


# Condition Variables

- The sender sends a notification.

| Member Function              | Description                  |
|------------------------------|------------------------------|
| <code>cv.notify_one()</code> | Notifies one waiting thread  |
| <code>cv.notify_all()</code> | Notifies all waiting threads |

- The receiver is waiting for the notification while holding the mutex.

| Member Function                                 | Description                                    |
|-------------------------------------------------|------------------------------------------------|
| <code>cv.wait(lock, ... )</code>                | Waits for the notification                     |
| <code>cv.wait_for(lock, relTime, ... )</code>   | Waits for the notification for a time duration |
| <code>cv.wait_until(lock, absTime, ... )</code> | Waits for the notification until a time point  |

 To protect against spurious wakeup and lost wakeup, the `wait` member function should be used with a predicate.

# Condition Variables

## Thread 1: Sender

- Prepares the work
- Notifies the receiver

```
// Prepares the work
{
    lock_guard<mutex> lck(mut);
    ready = true;
}
condVar.notify_one();
```

## Thread 2: Receiver

- Waits for its notification while holding the lock
  - Gets the lock
  - Checks and eventually continues to sleep
- Completes the work
- Releases the lock

```
{
    unique_lock<mutex>lck(mut);
    condVar.wait(lck, []{ return ready; });
    // Completes the work
} // Releases the lock
```

[conditionVariable.cpp](#)

# Performance Test: Ping Pong Game

- One thread executes a ping function, and the other 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.

| Execution Time      | Condition Variables | Atomic Flag | Atomic Bool | Semaphores |
|---------------------|---------------------|-------------|-------------|------------|
| Windows             | 0.7 sec             | 0.3 sec     | 0.4 sec     | 0.4 sec    |
| Linux (virtualized) | 21 sec              | 1.8 sec     | 2 sec       | 1.6 sec    |

[pingPongConditionVariable.cpp](#)

[pingPongAtomicFlag.cpp](#)

[pingPongAtomicBool.cpp](#)

[pingPongSemaphore.cpp](#)

# Latches and Barriers (C++20)

A thread waits at a synchronization point until the counter becomes zero.

- `latch` is useful for managing one task by multiple threads.

| Member Function                           | Description                                                                                        |
|-------------------------------------------|----------------------------------------------------------------------------------------------------|
| <code>lat.count_down(upd = 1)</code>      | Atomically decrements the counter by <code>upd</code> without blocking the caller.                 |
| <code>lat.try_wait()</code>               | Returns <code>true</code> if <code>counter == 0</code> .                                           |
| <code>lat.wait()</code>                   | Returns immediately if <code>counter == 0</code> . If not blocks until <code>counter == 0</code> . |
| <code>lat.arrive_and_wait(upd = 1)</code> | Equivalent to <code>count_down(upd); wait()</code> .                                               |

# Latches and Barriers (C++20)

- `barrier` is helpful to manage repetitive task through multiple threads.

| Member Function                    | Description                                                             |
|------------------------------------|-------------------------------------------------------------------------|
| <code>bar.arrive(upd = 1)</code>   | Atomically decrements counter by <code>upd</code> .                     |
| <code>bar.wait()</code>            | Blocks at the synchronization point until the completion step is done.  |
| <code>bar.arrive_and_wait()</code> | Equivalent to <code>arrive(); wait()</code> .                           |
| <code>bar.arrive_and_drop()</code> | Decrements the counter for the current and the subsequent phase by one. |

- The constructor gets a callable.
- In the completion phase, the callable is executed by an arbitrary thread.

[fullTimePartTimeWorkers.cpp](#)



# Cooperative Interruption (C++20)

Each running entity can be cooperatively interrupted.

- `std::jthread` and `std::condition_variable_any` support an explicit interface for the cooperative interruption.

Receiver (`std::stop_token token`)

| Member Function                     | Description                                                                                                                                      |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>token.stop_possible()</code>  | Returns <code>true</code> if <code>token</code> has an associated stop state.                                                                    |
| <code>token.stop_requested()</code> | <code>true</code> if <code>request_stop()</code> was called on the associated <code>std::stop_source src</code> , otherwise <code>false</code> . |

# Cooperative Interruption (C++20)

Sender (`std::stop_source`)

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

# Cooperative Interruption (C++20)

`std::stop_source` and `std::stop_token` are a general mechanism for sending a signal. They share a stop state.

➡ You can send a signal to any running entity.

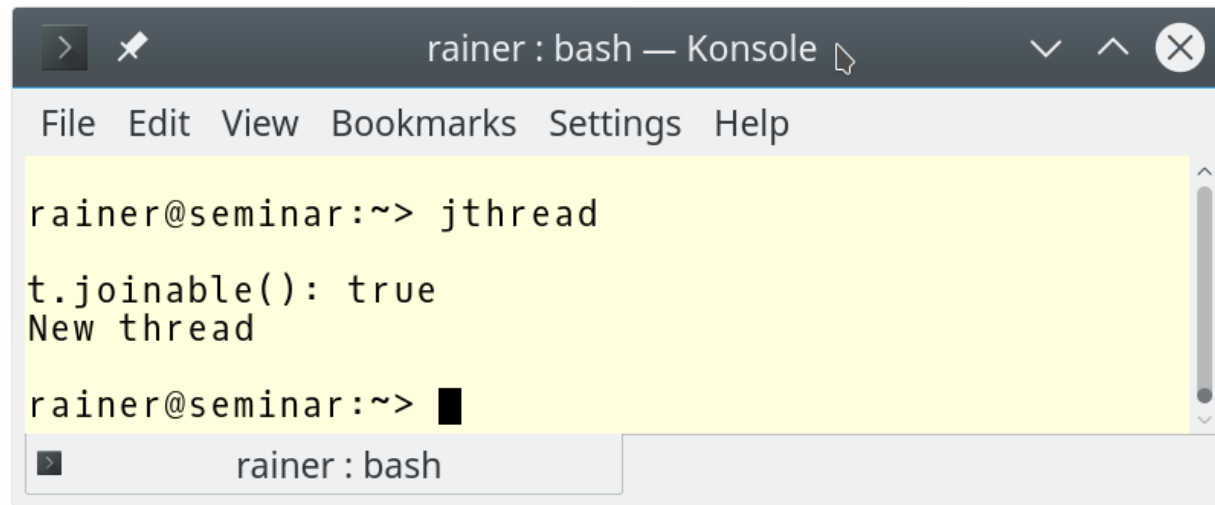
```
std::stop_source stopSource;  
std::stop_token stopToken = stopSource.get_token();  
  
void function(std::stop_token stopToken) {  
    if (stopToken.stop_requested()) return;  
}  
  
std::thread thr = std::thread(function, stopToken);  
stopSource.request_stop();
```

[signalStopRequests.cpp](#)

# std::jthread (C++20)

std::jthread joins automatically in its destructor.

```
std::jthread t{[] { std::cout << "New thread"; }};  
std::cout << "t.joinable(): " << t.joinable();
```



The screenshot shows a terminal window titled "rainer : bash — Konsole". The terminal output is as follows:

```
rainer@seminar:~> jthread  
t.joinable(): true  
New thread  
rainer@seminar:~> █
```

The terminal window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal text is highlighted in yellow.

[thread.cpp](#)  
[jthread.cpp](#)

# Synchronized Output Streams (C++20)

Synchronized output streams allow threads to write without interleaving on the same output stream.

- Predefined synchronized output streams

```
std::osyncstream for std::basic_osyncstream<char>
```

```
std::wosyncstream for std::basic_osyncstream<wchar_t>
```

- Synchronized output streams

- Output is written to the internal buffer of type `std::basic_syncbuf`
- When the output stream goes out of scope, it outputs its internal buffer

# Synchronized Output Streams (C++20)

- Permanent variable `synced_out`

```
{  
    std::osyncstream synced_out(std::cout);  
    synced_out << "Hello, ";  
    synced_out << "World!";  
    synced_out << std::endl; // no effect  
    synced_out << "and more!\n";  
} // destroys the synced_output and emits the internal buffer
```

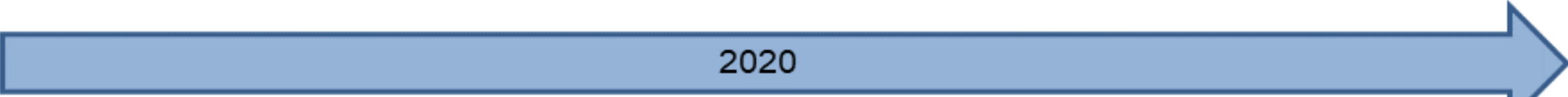
- Temporary Variable

```
std::osyncstream(std::cout) << "Hello, " << "World!\n";
```

[sequencedOutput.cpp](#)

# C++20 - Concurrency

2020



## The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

## Core Language

- Three-way comparison operator
- Designated initialization
- `constexpr` and `constinit`
- Template improvements
- Lambda improvements
- New attributes

## Library

- `std::span`
- Container improvements
- Arithmetic utilities
- Calendar and time zone
- Formatting library

## Concurrency

- Atomics
- Semaphores
- Latches and barriers
- Cooperative interruption
- `std::jthread`



Blog: [www.ModernesCpp.com](http://www.ModernesCpp.com)

Mentoring: [www.ModernesCpp.org](http://www.ModernesCpp.org)

Rainer Grimm

Training, Mentoring, and  
Technology Consulting