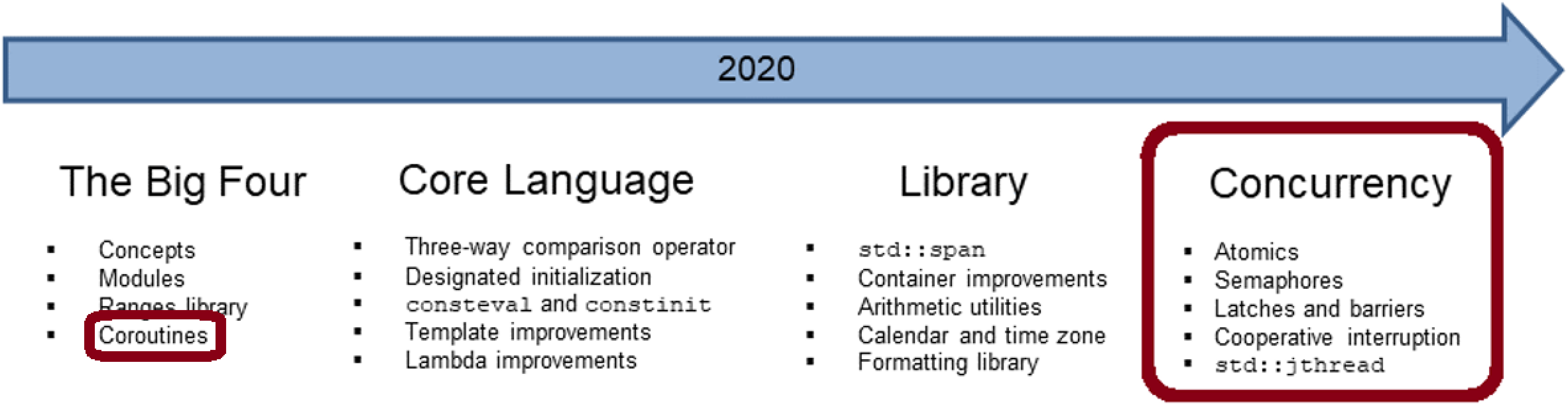
Concurrency in Modern C++

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
- `consteval` and `constinit`
- Template improvements
- Lambda improvements

Library

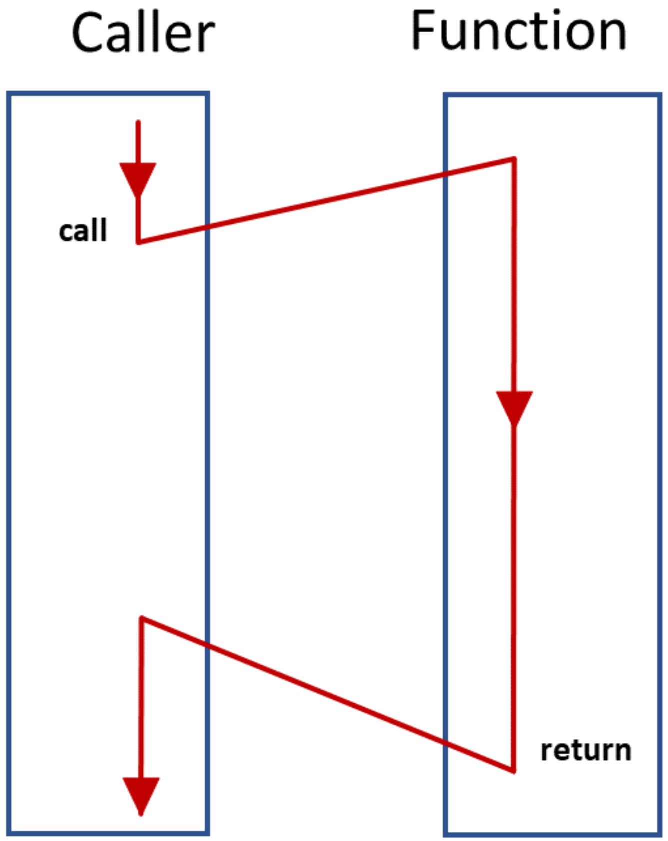
- `std::span`
- Container improvements
- Arithmetic utilities
- Calendar and time zone
- Formatting library

Concurrency

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

Coroutines

Coroutines are generalized functions that can be paused and resumed while saving their state.

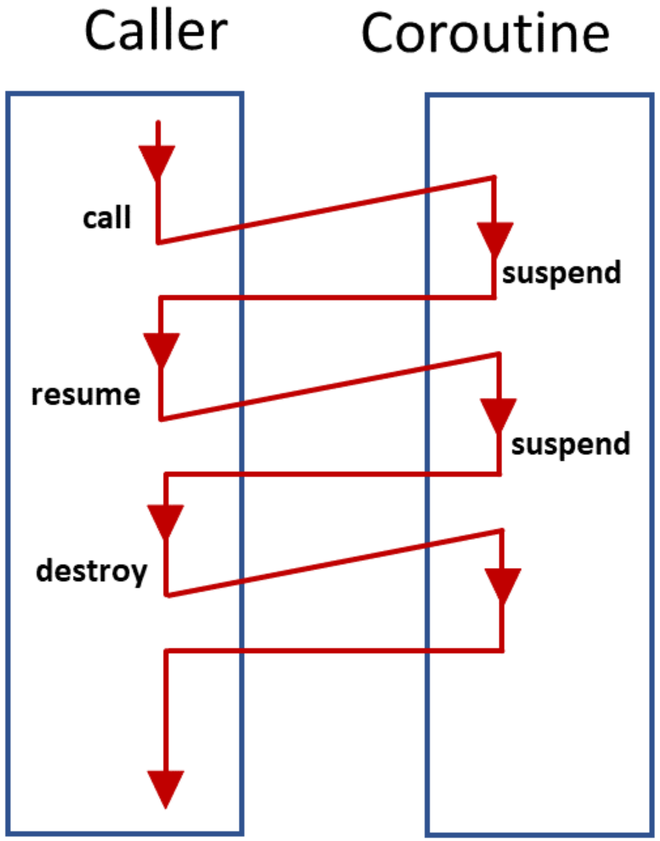


Caller

Function

call

return



Caller

Coroutine

call

suspend

resume

suspend

destroy

Characteristics

- Two new concepts
  - `co_await expression`: suspend and resume expression
  - `co_yield expression`: supports generators

- Typical use cases
  - Cooperative Tasks
  - Event loops
  - Infinite data streams
  - Pipelines

Characteristics

Design Principles (James McNellis)

- **Scalable**, to billions of concurrent coroutines
- **Efficient**: Suspend/resume operations comparable in cost to function call overhead
- **Open-Ended**: Library designers can develop coroutine libraries
- **Seamless Interaction** with existing facilities with no overhead.
- **Usable** in environments where exceptions are forbidden or not available.

Characteristics

	Function	Coroutine
`invoke`	`func(args)`	`func(args)`
`return`	`return statement`	`co_return statement`
`suspend`		`co_await expression` `co_yield expression`
`resume`		`coroutine_handle<>::resume()`

A function is a coroutine if it contains a call `co_return`, `co_await`, `co_yield`, or a range-based for loop `co_await`.

Coroutines: Generators

```
Generator<int> getNext(int start = 0, int step = 1) {
    auto value = start;
    while (true) {
        co_yield value;
        value += step;
    }
}
...
auto gen = getNext(-10);
for (int i= 1; i <= 20; ++i) std::cout << gen.next() << " ";
```

➡

```
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
```

infiniteDataStreamWithComments.cpp

Coroutines: Generators

C++23 has the first concrete coroutine `std::generator`

- `std::generator`
  - Generates a sequence of elements
  - Enables nested calls of generators

```
std::generator<int> fib() {
  co_yield 0;
  auto a = 0;
  auto b = 1;
  for(auto n : std::views::iota(0)) {
    auto next = a + b;
    a = b;
    b = next;
    co_yield next;
  }
}
```

```
std::generator<int> getGenerator() {
    co_yield fib();
}
```

```
std::generator<int> getElements() {
    co_yield std::ranges::elements_of(fib());
}
```

➡ `getGenerator`: returns the generator
`getElements`: returns the next element

Coroutines: Waiting Instead of Blocking

Blocking

```
Acceptor accept{443};

while (true){
    Socket so= accept.accept(); // block
    auto req= so.read();        // block
    auto resp= handleRequest(req);
    so.write(resp);             // block
}
```

Waiting

```
Acceptor accept{443};

while (true){
    Socket so= co_await accept.accept();
    auto req= co_await so.read();
    auto resp= handleRequest(req);
    co_await so.write(resp);
}
```

Framework

C++20 offers a framework for creating concrete coroutines.

```
auto gen = coroutineFactory();
gen.next();
auto result = gen.getValue();
```

- The framework consists of three components:
    - The promise object
    - The coroutine handle
    - The coroutine frame

Framework

The **promise object** needs the following member functions.

Member Functions	**Description**
Default constructor	
`initial_suspend()`	Determines if the coroutine suspends before it runs.
`final_suspend()`	Determines if the coroutine suspends before it ends.
`unhandled_exception()`	Called when an exception happens.
`get_return_object()`	Returns the coroutine object (resumable object).
`return_value(val)`	Is invoked by `co_return val`.
`return_void`	Is invoked by `co_return`.
`yield_value(val)`	Is invoked by `co_yield val`.

Framework

The **coroutine handle** is a non-owning handle to resume or destroy the coroutine frame from the outside.

The **coroutine frame**

- Heap allocated
- Consists of
  - Promise object
  - Coroutine parameters
  - Representation of the suspension point
  - Local variables

Awaitables and Awaiters

The three promise functions `yield_value`, `inital_suspend`, and `final_suspend` return Awaiters.

- An Awaiter
  - Is something you can await on
  - Has to support three functions

Function	**Description**
`await_ready`	Indicates if the result is ready. When it returns `false`, `await_suspend` is called.
`await_suspend`	Schedule the coroutine for resumption or destruction.
`await_resume`	Provides the result for the `co__await expr` expression.

Two Predefined Awaiters

- **`std::suspend_always`**

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

- **`std::suspend_never`**

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

Awaiters

- Steps to get the Awaiter
  - Look for the `co_await operator` in the promise object
    - `awaiter = awaitable.operator co_await();`
  - Look for a freestanding `co_await operator`
    - `awaiter = operator co_await();`
  - The Awaitable becomes the Awaiter
    - `awaiter = awaitable;`

The Promise Workflow

The compiler transforms a coroutine into the following workflow.

```
{
    Promise prom;
    co_await prom.initial_suspend();
    try {
        <function body having co_return, co_yield, or co_wait>
    }
    catch (...) {
        prom.unhandled_exception();
    }
FinalSuspend:
    co_await prom.final_suspend();
}
```

lazyFutureWithComments.cpp

The Awaiter Workflow

The compiler creates the following workflow based on the Awaitable.

```
awaitable.await_ready() returns false:
    suspend coroutine
    awaitable.await_suspend(coroutineHandle) returns:
        void:
        bool:
        another coroutine handle:
resumptionPoint:
return awaitable.await_resume();
```

Return value of `awaitable.await_suspend()`	Description
`void`	The coroutine keeps suspended and returns to the caller.
`bool == true`	The coroutine keeps suspended and returns to the caller.
`bool == false`	The coroutine is resumed and does not return to the caller.
`anotherCoroutineHandle`	The other coroutine is resumed and returns to the caller.

`startJobWithAutomaticResumptionOnThread.cpp`

Atomics

Atomics are the foundation of C++ memory model

➡ Atomic 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 structure guaranteed to be lock free.

- `std::atomic`

```
std::atomic<T*>
std::atomic<Integral type>
std::atomic<User-defined type>
std::atomic<floating point> (C++20)
std::atomic<smart pointers> (C++20)
```

Atomics

Operation	Description
`test_and set`	Sets the value and returns the previous value.
`clear`	Clears the value
`is_lock_free`	Checks if the atomic object is lock-free.
`load`	Returns the value of the atomic.
`store`	Replaces the value of the atomic with the non-atomic.
`exchange`	Replaces the value with the new value. Returns the old value.
`compare_exchange_weak` `compare_exchange_strong`	`atom.compare_exchange_strong(expect, desir)` ▪ If `atom` is equal to `expect` returns `true`, `atom` becomes `desir`. ▪ If not returns `false`, `expect` is updated with `atom`.
`fetch_add, +=` `fetch_sub, -=`	Adds (substrct) the value and returns the preious value.
`++, --`	Increments or decrements the atomic.

`fetch_mult.cpp`

Atomics

- `std::atomic_flag` and `std::atomic` (C++20)
  - Enable synchronization of threads
    - `atom.notify_one()`: Notifies one waiting operation
    - `atom.notify_all()`: Notifies all waiting operations
    - `atom.wait(val)`: Waiting for a notification and blocks as long as `atom == val` holds
  - The default constructor initializes the value.

`atomicWaitAtomicBool.cpp`

Atomics

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

- General rule: use smart pointer
- But:
  - The handling of the control block is thread-safe.
  - Access to the resource is not thread-safe.
- Solution:
  - `std::atomic_shared_ptr`
  - `std::atomic_weak_ptr`

Atomics

3 reasons for an atomic smart pointer.

- 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 general use

Atomics

`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 such as `std::atomic`

- `std::atomic_ref`

```
std::atomic_ref<T*>
std::atomic_ref<Integral type>
std::atomic_ref<User-defined type>
std::atomic_ref<floating point>
```

[atomicReference.cpp]()

Semaphores

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

Member Function	**Description**
`counting_semaphore::max()`	Returns the maximum value of the `counter.`
`sem.release(upd = 1)`	Increases the counter atomically by `upd` and unblocks threads acquiring the semaphore
`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.

[`threadSynchronisationSemaphore.cpp`]()

Condition Variables

- The sender sends a notification.

Member Function	Description
`cv.notify_one()`	Notifies one waiting thread
`cv.notify_all()`	Notifies all waiting threads

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

Member Function	Description
`cv.wait(lock, ... )`	Waits for the notification
`cv.wait_for(lock, relTime, ... )`	Waits for the notification for a time period
`cv.wait_until(lock, absTime, ... )`	Waits for the notification until a time point

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

Condition Variables

Thread 1: Sender

- Does its work
- Notifies the receiver

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

conditionVariable.cpp

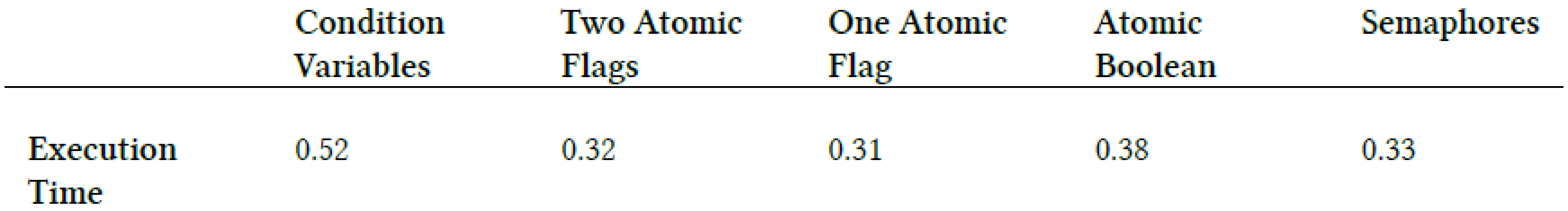
Thread 2: Receiver

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

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

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.



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

pingPongConditionVariable.cpp
pingPongAtomicTwoFlags.cpp
pingPongAtomicOneFlag.cpp
pingPongAtomicBool.cpp
pingPongSemaphore.cpp

Latches and Barriers

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**
`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();`

[`workers.cpp`](workers.cpp)

Latches and Barriers

- `barrier` is helpful for managing repeated tasks by multiple threads.

Member Function	**Description**
`bar.arrive(upd = 1)`	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.

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

[`fullTimePartTimeWorkers.cpp`](fullTimePartTimeWorkers.cpp)

Cooperative Interruption

Each running entity can be cooperative interrupted.

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

Receiver (`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`.

Cooperative Interruption

Sender (`std::stop_source`)

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.

[`interruptJthread.cpp`]

Cooperative Interruption

`std::stop_source` and `std::stop_token` are a general mechanism to send a signal.

⇨ You can send a signal to any running entity.

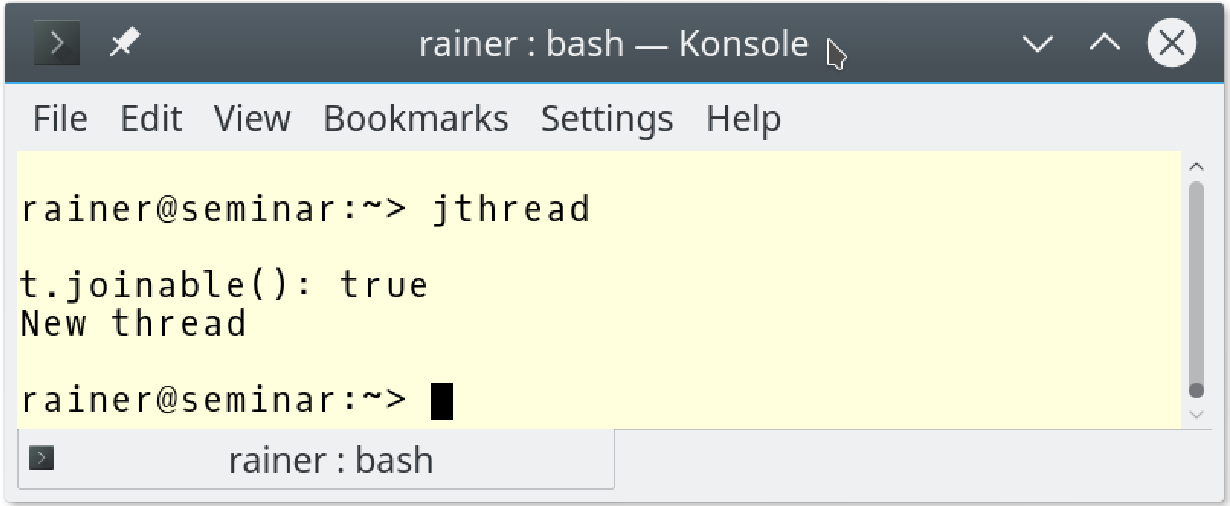
```
std::stop_source stopSource;
std::stop_token stopToken = stopSource.get_token();

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

`std::jthread` joines automatically in its destructor.

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



```
rainer@seminar:~> jthread

t.joinable(): true
New thread

rainer@seminar:~>
```

thread.cpp
jthread.cpp

Synchronized Output Streams

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

- 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]()

Blog: www.ModernesCpp.com

Mentoring: www.ModernesCpp.org

Rainer Grimm

Training, Mentoring, and
Technology Consulting