

```
#include <iostream>
```

```
int main(){
```

# Concurrency and Parallelism with C++17 and C++20/23

```
std::cout <<
```

```
std::vector<int>
```

```
std::iota(myVec.begin(), myVec.end(), 0);
```

```
std::cout <<
```

```
for ( auto i: myVec) std::cout <<
```

```
std::cout <<
```

```
std::function< bool(int)> myBindP
```

```
myVec.erase(std::remove_if(myVec.begin(), myVec.end(), myBindP
```

```
std::cout << "myVec2: "
```

```
for ( auto i: myVec2) std::cout <<
```

```
std::cout << "\n\n";
```

```
std::vector<int> myVec2(20);
```

```
std::iota(myVec2.begin(), myVec2.end(), 0);
```

```
std::cout << "myVec2: "
```

```
for ( auto i: myVec2) std::cout <<
```

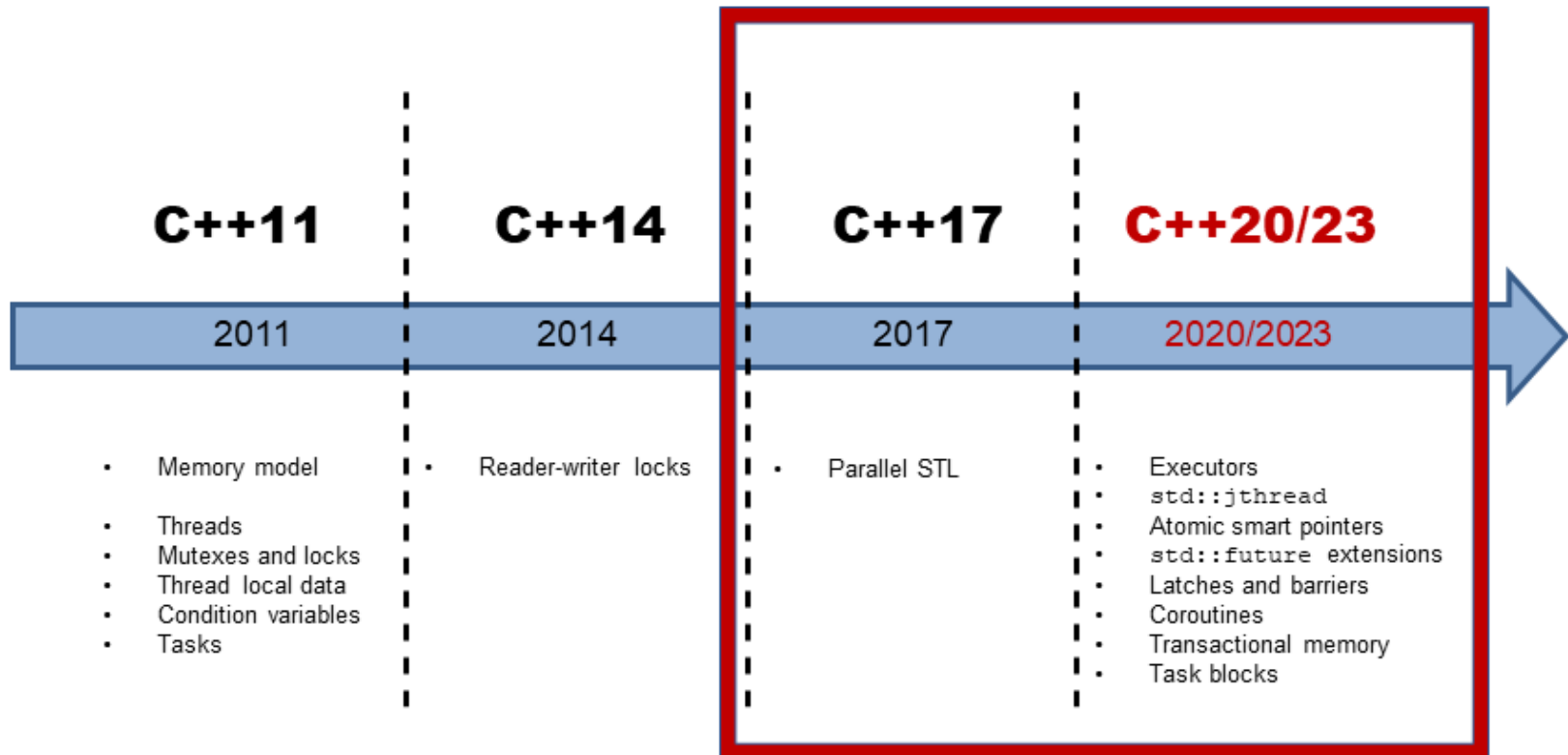
```
std::cout <<
```

Rainer Grimm

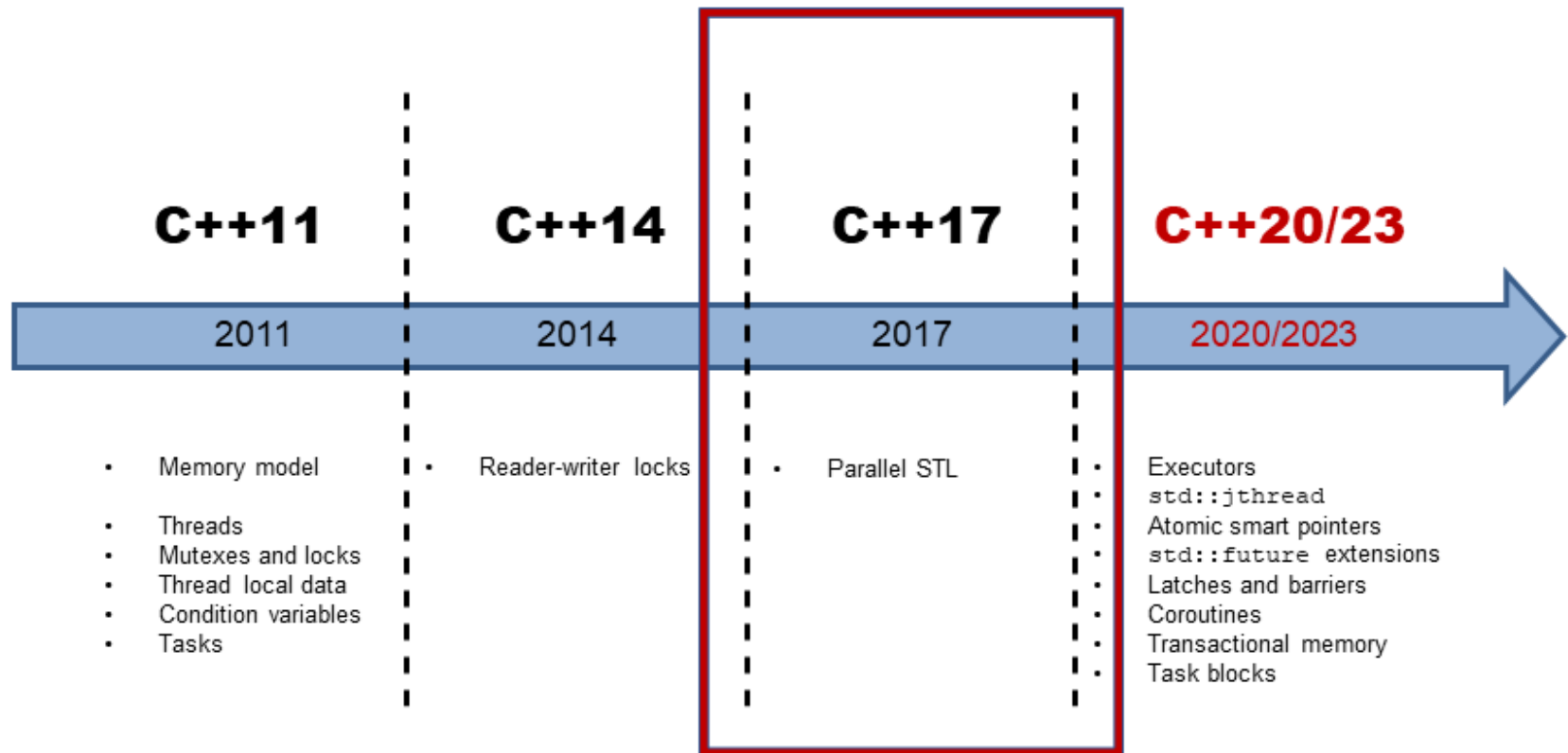
Training, Coaching and,  
Technology Consulting

[www.ModernesCpp.de](http://www.ModernesCpp.de)

# Concurrency and Parallelism in C++



# Concurrency and Parallelism in C++17



# Parallel STL

You can choose the execution policy of an algorithm.

- Execution policies

- `std::execution::seq`

- Sequential in one thread

- `std::execution::par`

- Parallel

- `std::execution::par_unseq`

- Parallel and vectorised  SIMD

# Parallel STL

```
const int SIZE = 8;
int vec[]={1, 2 , 3, 4, 5, 6, 7, 8};
int res[SIZE] = {0,};

int main(){
    for (int i= 0; i < SIZE; ++i){
        res[i] = vec[i] + 5;
    }
}
```

## Not vectorised

```
movslq -8(%rbp), %rax
movl   vec(,%rax,4), %ecx
addl   $5, %ecx
movslq -8(%rbp), %rax
movl   %ecx, res(,%rax,4)
```

## Vectorised

```
movdqa .LCPI0_0(%rip), %xmm0    # xmm0 = [5,5,5,5]
movdqa vec(%rip), %xmm1
padd   %xmm0, %xmm1
movdqa %xmm1, res(%rip)
padd   vec+16(%rip), %xmm0
movdqa %xmm0, res+16(%rip)
xorl   %eax, %eax
```

# Parallel STL

```
using namespace std;
vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

sort(vec.begin(), vec.end());           // sequential as ever

sort(execution::seq, vec.begin(), vec.end());           // sequential
sort(execution::par, vec.begin(), vec.end());           // parallel
sort(execution::par_unseq, vec.begin(), vec.end());     // par + vec
```

# Parallel STL

adjacent\_difference, adjacent\_find, all\_of, any\_of, copy,  
copy\_if, copy\_n, count, count\_if, equal, **exclusive\_scan**,  
fill, fill\_n, find, find\_end, find\_first\_of, find\_if,  
find\_if\_not, **for\_each**, **for\_each\_n**, generate, generate\_n,  
includes, **inclusive\_scan**, inner\_product, inplace\_merge,  
is\_heap, is\_heap\_until, is\_partitioned, is\_sorted,  
is\_sorted\_until, lexicographical\_compare, max\_element,  
merge, min\_element, minmax\_element, mismatch, move,  
none\_of, nth\_element, partial\_sort, partial\_sort\_copy,  
partition, partition\_copy, **reduce**, remove, remove\_copy,  
remove\_copy\_if, remove\_if, replace, replace\_copy,  
replace\_copy\_if, replace\_if, reverse, reverse\_copy,  
rotate, rotate\_copy, search, search\_n, set\_difference,  
set\_intersection, set\_symmetric\_difference, set\_union,  
sort, stable\_partition, stable\_sort, swap\_ranges,  
transform, **transform\_exclusive\_scan**,  
**transform\_inclusive\_scan**, **transform\_reduce**,  
uninitialized\_copy, uninitialized\_copy\_n,  
uninitialized\_fill, uninitialized\_fill\_n, unique,  
unique\_copy

# Parallel STL

`std::transform_reduce`

- Haskell's function `map` is called `std::transform` in C++
- `std::transform_reduce`  `std::map_reduce`

```
std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};

std::size_t res = std::transform_reduce(std::execution::par,
                                         strVec.begin(), strVec.end(), 0,
                                         [](std::size_t a, std::size_t b){ return a + b; },
                                         [](std::string s){ return s.length(); });

std::cout << res;    // 21
```

# Parallel STL

- Danger of data races or deadlocks

```
int numComp = 0;
std::vector<int> vec = {1, 3, 8, 9, 10};
std::sort(std::execution::par, vec.begin(), vec.end(),
          [&numComp](int fir, int sec){ numComp++; return fir < sec; }
);
```

➡ The access to **numComp** has to be atomic.

# Parallel STL

- Support for `std::execution::par`

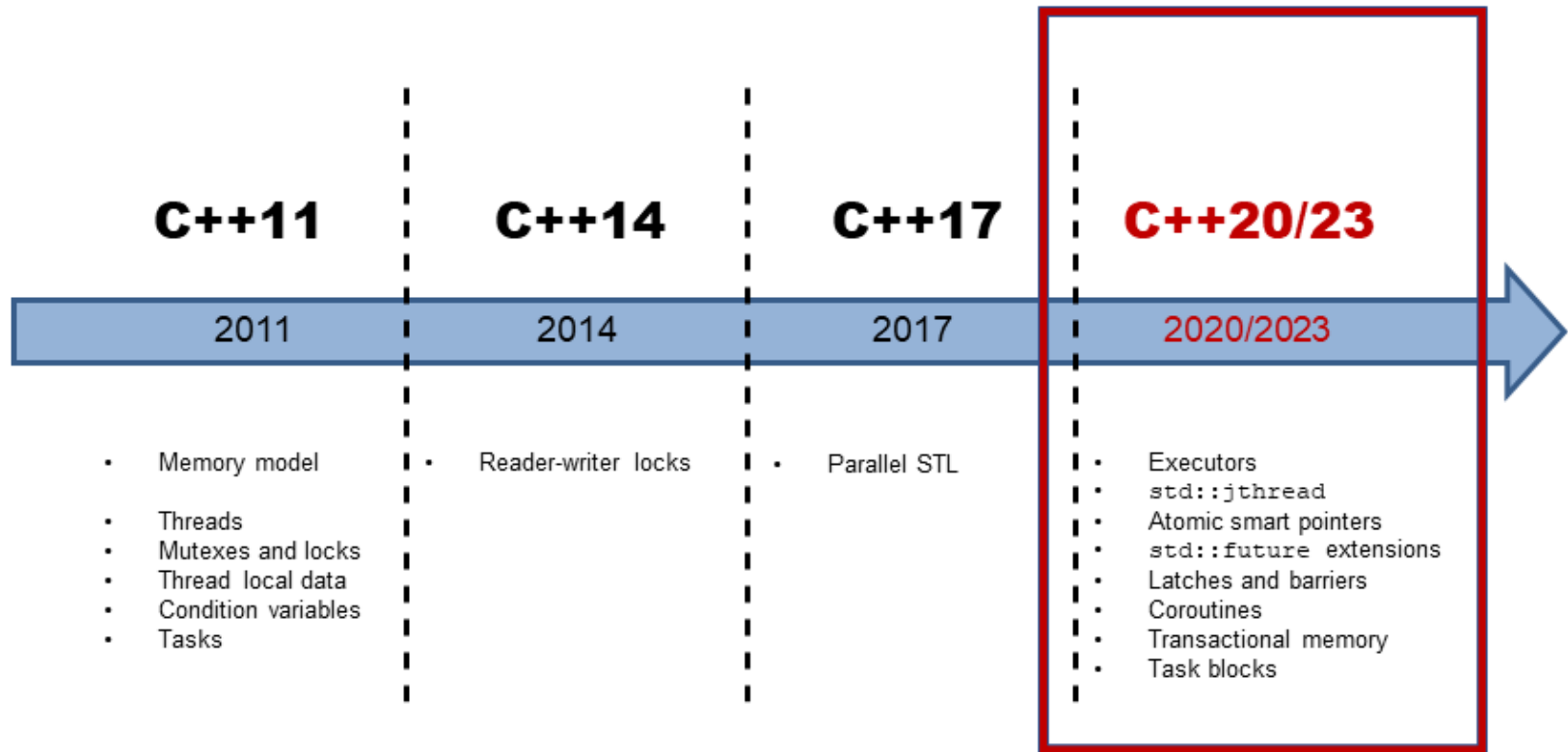
- ~~GCC and Clang~~

- MSVC with Visual Studio 2017 15.8 (/std=c++latest)

`copy, copy_n, fill, fill_n, move, reverse,`  
`reverse_copy, rotate, rotate_copy, swap_ranges`

`adjacent_difference, adjacent_find, all_of, any_of,`  
`count, count_if, equal, exclusive_scan, find,`  
`find_end, find_first_of, find_if, for_each,`  
`for_each_n, inclusive_scan, mismatch, none_of,`  
`partition, reduce, remove, remove_if, search,`  
`search_n, sort, stable_sort, transform,`  
`transform_exclusive_scan, transform_inclusive_scan,`  
`transform_reduce`

# Concurrency and Parallelism in C++20/23



# Executors

Executors are the basic building block for execution in C++.

- They fulfil a similar role for execution such as allocators for allocation.

An executor consists of a set of rules for a callables:

- **Where:** run on a internal or external processor
- **When:** run immediately or later
- **How:** run on a CPU or GPU

# Executors

- Using an executor

```
my_executor_type my_executor = ... ;  
auto future = std::async(my_executor, []{  
    std::cout << "Hello world " << std::endl; }  
);
```

```
std::for_each(std::execution::par.on(my_executor),  
    data.begin(), data.end(), func);
```

- Obtaining an executor

```
static_thread_pool pool(4);  
auto exec = pool.executor();  
task1 = long_running_task(exec);
```

# Executors

An executor provides one or more execution functions for creating a callable.

| Name                | Cardinality | Direction | Standard               |
|---------------------|-------------|-----------|------------------------|
| execute             | single      | oneway    | <del>C++20</del> C++23 |
| twoway_execute      | single      | twoway    | C++23                  |
| then_execute        | single      | then      | C++23                  |
| bulk_execute        | bulk        | oneway    | <del>C++20</del> C++23 |
| bulk_twoway_execute | bulk        | twoway    | C++23                  |
| bulk_then_execute   | bulk        | then      | C++23                  |

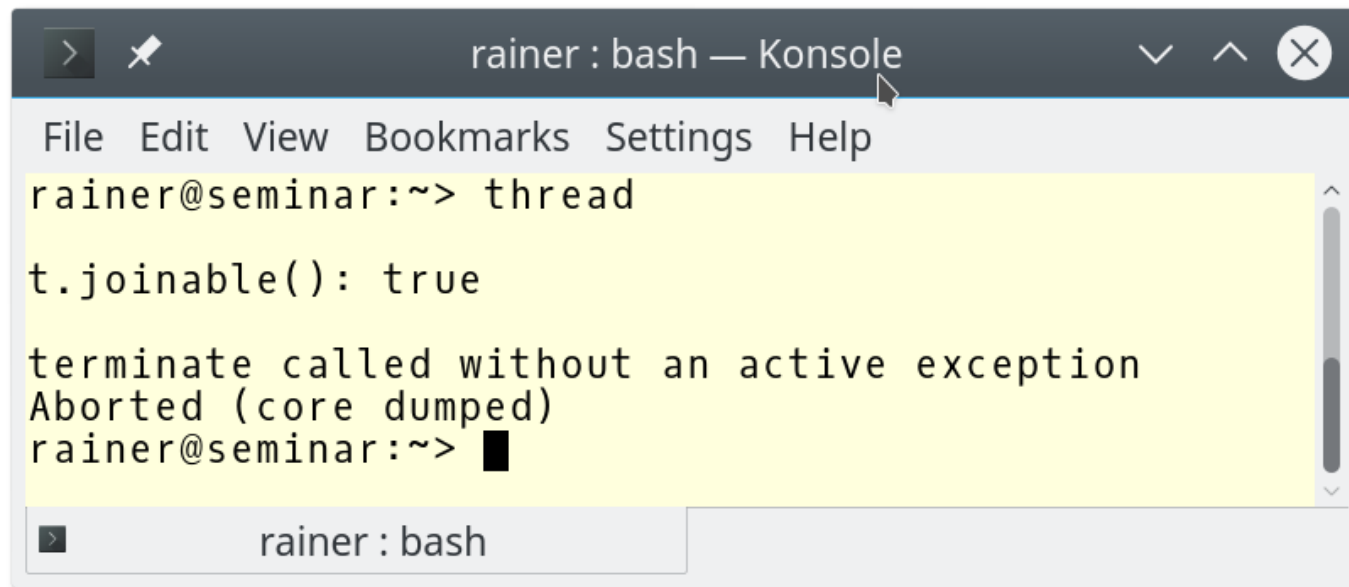
**Cardinality:** Creation of one execution agent or a group of execution agents.

**Direction:** Directions of the execution.

# std::jthread

**Problem:** `std::thread` throws `std::terminate` in its destructor if still joinable.

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



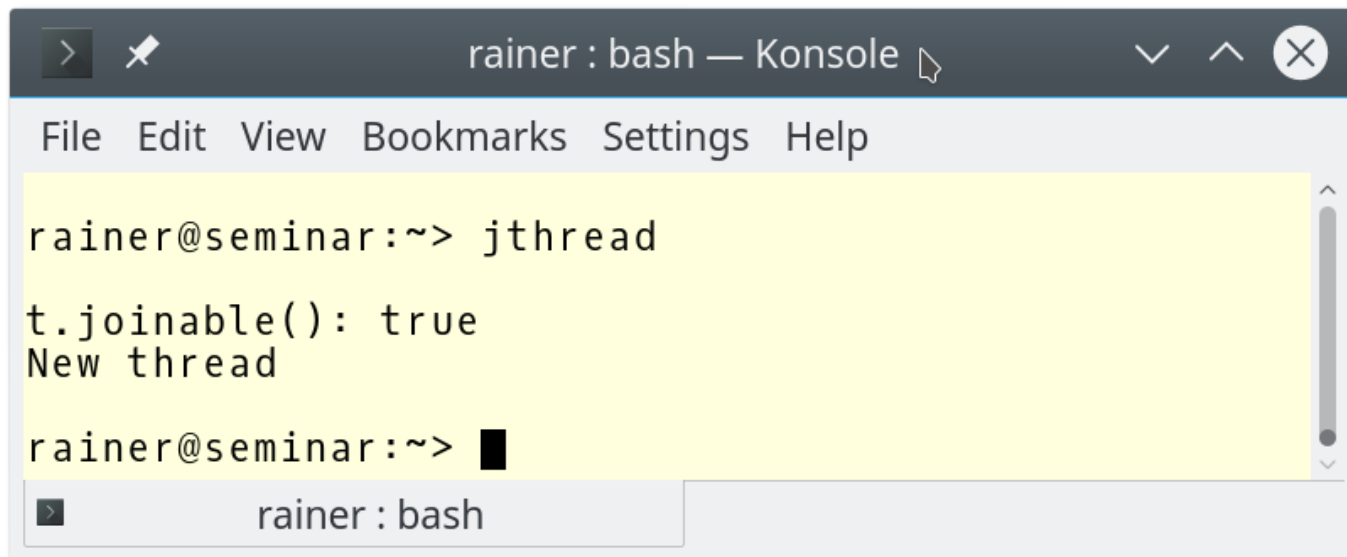
```
rainer : bash — Konsole  
File Edit View Bookmarks Settings Help  
rainer@seminar:~> thread  
t.joinable(): true  
terminate called without an active exception  
Aborted (core dumped)  
rainer@seminar:~> █
```

The screenshot shows a terminal window titled "rainer : bash — Konsole". The window contains the output of a C++ program. The program creates a thread, prints "t.joinable(): true", and then calls `std::terminate`, which results in an "Aborted (core dumped)" message. The terminal prompt is `rainer@seminar:~>`.

# `std::jthread`

**Solution:** `std::jthread` joins automatically at the end of its scope.

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



A screenshot of a terminal window titled "rainer : bash — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows the following sequence of commands and output:

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

The terminal window has a tab at the bottom labeled "rainer : bash".

# `std::jthread`

- Instances of `std::jthread` can be interrupted

## Receiver (`stop_token`)

- Explicit check:
  - `stop_requested`: yields, when an interrupt was signalled
  - `std::condition_variable_any` **wait** variations with predicator
- Implicit check:
  - Register a callback via `std::stop_callback`

## Sender (`stop_source`)

- `request_stop`: signals an interrupt

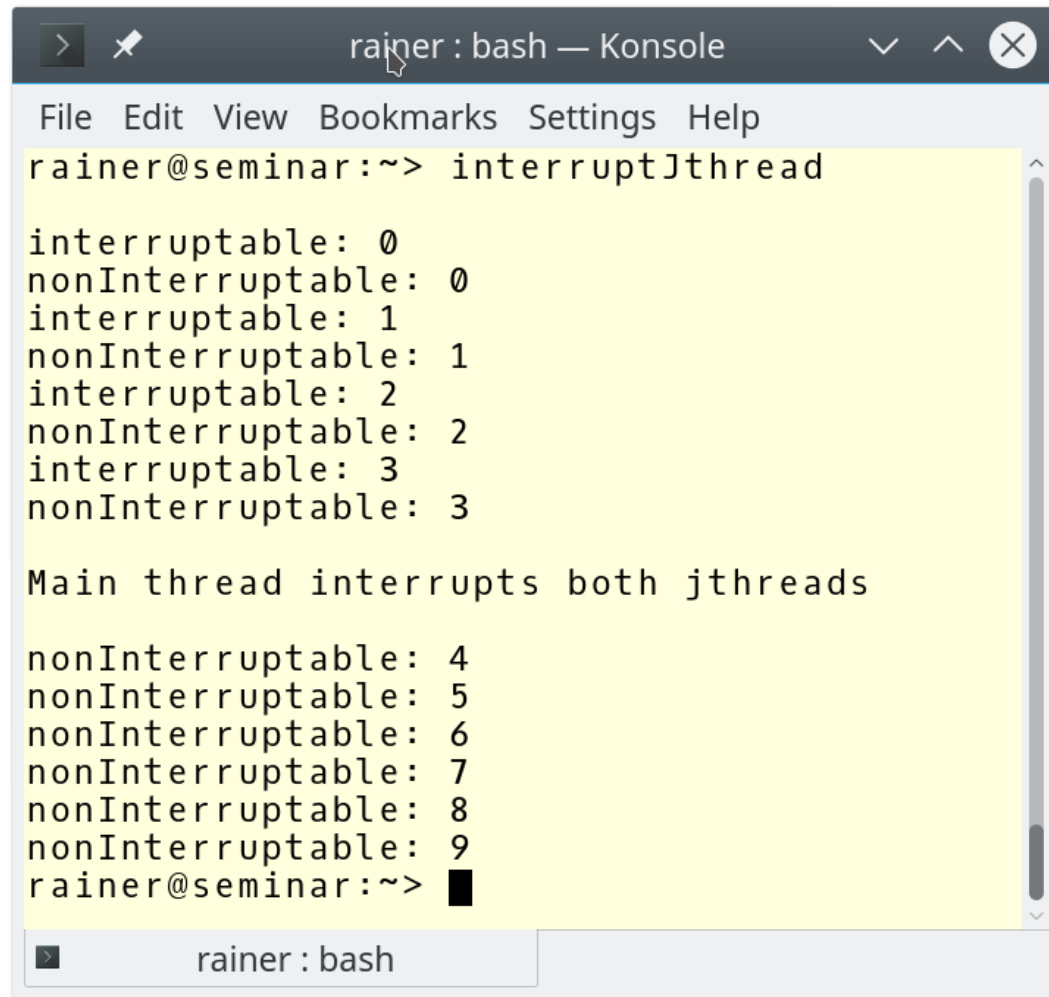
# std::jthread

```
jthread nonInterruptable([]{  
    int counter{0};  
    while (counter < 10){  
        this_thread::sleep_for(0.2s);  
        cerr << "nonInterruptable: "  
            << counter << endl;  
        ++counter;  
    }  
});
```

```
jthread interruptable([](stop_token token){  
    int counter{0};  
    while (counter < 10){  
        this_thread::sleep_for(0.2s);  
        if (token.stop_requested()) return;  
        cerr << "interruptable: "  
            << counter << endl;  
        ++counter;  
    }  
});
```

```
this_thread::sleep_for(1s);  
cerr << endl;  
cerr << "Main thread interrupts both jthreads" << endl;  
nonInterruptable.request_stop();  
interruptable.request_stop();  
cout << endl;
```

# std::jthread



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> interruptJthread

interruptable: 0
nonInterruptable: 0
interruptable: 1
nonInterruptable: 1
interruptable: 2
nonInterruptable: 2
interruptable: 3
nonInterruptable: 3

Main thread interrupts both jthreads

nonInterruptable: 4
nonInterruptable: 5
nonInterruptable: 6
nonInterruptable: 7
nonInterruptable: 8
nonInterruptable: 9
rainer@seminar:~>
```

# Atomic Smart Pointers

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

- General Rule:
  - You should use smart pointers.
- But:
  - The managing of the control block and the deletion of the resource is thread-safe. The access to the resource is not thread-safe.
  - ➡ **Tony Van Eerd: Forget what you learned in Kindergarten. Stop sharing.**
- *Solution:*
  - `std::atomic_shared_ptr`
  - `std::atomic_weak_ptr`

# Atomic Smart Pointer

## 3 Reasons

- Consistency
  - `std::shared_ptr` is the only non-atomic data type for which atomic operations exists.
- Correctness
  - The correct usage of atomic operations is just based on the discipline of the user.  extremely error-prone
  - `std::atomic_store(&sharPtr, localPtr)  sharPtr = localPtr`
- Performance
  - `std::shared_ptr` has to be design for the special use-case.

# Atomic Smart Pointer

```
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);
    }
};
```

# Atomic Smart Pointers

Atomic smart pointers are part of the ISO C++20 standard.

- Partial specialisation of `std::atomic`
- `std::atomic_shared_ptr`  
➡ `std::atomic<std::shared_ptr<T>>`
- `std::atomic_weak_ptr`  
➡ `std::atomic<std::weak_ptr<T>>`

# std::future Extensions

std::future doesn't support composition

- std::future Improvement ➡ Continuation
  - then: execute the next future if the previous one is done

```
future<int> f1 = async([](){ return 123; });
future<string> f2 = f1.then([](future<int> f){
    return to_string(f.get());           // non-blocking
});
auto myResult = f2.get();                // blocking
```

# std::future Extensions

- **when\_all**: execute the future if all futures are done

```
future<int> futures[] = { async([]() { return intResult(125); }),  
                        async([]() { return intResult(456); }) };  
future<vector<future<int>>> all_f = when_all(begin(futures), end(futures));  
  
vector<future<int>> myResult = all_f.get();  
  
for (auto fut: myResult): fut.get();
```

- **when\_any**: execute the future if one of the futures is done

```
future<int> futures[] = { async([]() { return intResult(125); }),  
                        async([]() { return intResult(456); }) };  
when_any_result<vector<future<int>>> any_f = when_any(begin(futures),  
                                                    end(futures));  
  
future<int> myResult = any_f.futures[any_f.index];  
  
auto myResult = myResult.get();
```

# std::future Extensions

- **make\_ready\_future** and **make\_exception\_future**: create a future directly

```
future<int> compute(int x){  
    if (x < 0) return make_ready_future<int>(-1);  
    if (x == 0) return make_ready_future<int>(0);  
    future<int> f1 = async([]{ return do_work(x); });  
    return f1;  
}
```



## Further information

[C++17: I See a Monad in Your Future!](#) (Bartosz Milewski)

# std::future Extensions

- Disadvantages of the extended futures

- Futures and promises are coupled to `std::thread`.
- Where is the `.then` continuation be invoked?
- Passing futures to `.then` continuation is too verbose.

```
std::future f1 = std::async([]{ return 123; });  
std::future f2 = f1.then([](std::future f){  
    return std::to_string(f.get());  
});
```

```
std::future f2 = f1.then(std::to_string);
```

- Future blocks in its destructor.
- Futures und values should be easily composable.

```
bool f(std::string, double, int);  
std::future<std::string> a = /* ... */;  
std::future<int> c = /* ... */;  
f(a.get(), 3.14, c.get())
```

```
std::future<bool> d2 = when_all(a, 3.14, c).then(f);
```

# Latches and Barriers

C++ has no semaphore → latches and barriers

- Key idea

A thread is waiting at the synchronisation point until the counter becomes zero.

- `latch` is for the one-time use-case

- `count_down_and_wait`: decrements the counter until it becomes zero
- `count_down(n = 1)`: decrements the counter by `n`
- `is_ready`: checks the counter
- `wait`: waits until the counter becomes zero

# Latches and Barriers

- `barrier` can be reused
  - `arrive_and_wait`: waits at the synchronisation point
  - `arrive_and_drop`: removes itself from the synchronisation mechanism
- `flex_barrier` is a reusable and adaptable barrier
  - The constructor gets a callable.
  - The callable will be called in the completion phase.
  - The callable returns a number which stands for the counter in the next iteration.
  - Can change the value of the counter for each iteration.

# Latches and Barriers

```
void doWork(threadpool* pool) {  
    latch completion_latch(NUMBER_TASKS);  
    for (int i = 0; i < NUMBER_TASKS; ++i) {  
        pool->add_task([&] {  
            // perform the work  
            ...  
            completion_latch.count_down();  
        });  
    }  
    // block until all tasks are done  
    completion_latch.wait();  
}
```

# Coroutines

Coroutines are generalised functions that can be suspended and resumed while keeping their state.

- Typical use-case
  - Cooperative Tasks (protection from data races)
  - Event loops
  - Infinite data streams
  - Pipelines

# Coroutines

## Design Principles

- **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.

# Coroutines

|         | Function                      | Coroutine                                                              |
|---------|-------------------------------|------------------------------------------------------------------------|
| invoke  | <code>func(args)</code>       | <code>func(args)</code>                                                |
| return  | <code>return statement</code> | <code>co_return someValue</code>                                       |
| suspend |                               | <code>co_await someAwaitable</code><br><code>co_yield someValue</code> |
| resume  |                               | <code>coroutine_handle&lt;&gt;::resume()</code>                        |

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

# Coroutines

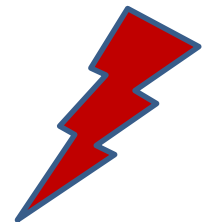
```
generator<int> genForNumbers(int begin, int inc = 1){  
    for (int i = begin; ; i += inc){  
        co_yield i;  
    }  
}
```

```
int main(){  
    auto numbers = genForNumbers(-10);  
    for (int i = 1; i <= 20; ++i) std::cout << numbers << " ";  
    for (auto n: genForNumbers(0, 5)) std::cout << n << " ";  
}
```



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

**0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 ....**



# Coroutines

## 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);
}
```

# Transactional Memory

*Transactional Memory* is the idea of transactions from the data base theory applied to software.

- A transaction has the ACID properties without ***Durability***

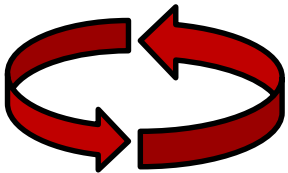
```
atomic{  
    statement1;  
    statement2;  
    statement3;  
}
```

- **Atomicity:** all or no statement will be performed
- **Consistency:** the system is always in a consistent state
- **Isolation:** a transaction runs total isolation
- **Durability:** the result of a transaction will be stored

# Transactional Memory

- Transactions
  - build a total order
  - feel like a global lock
    - ➔ Optimistic approach  $\neq$  lock
- Workflow

## *Retry*



A transaction stores its initial state.

The transaction will be performed without synchronisation.

The runtime experiences a violation to the initial state.

The transaction will be performed once more.

## *Rollback*



# Transactional Memory

- Two forms

- `synchronized blocks`

- *relaxed* transaction

- are not transaction in the pure sense

- ➔ can have `transaction-unsafe` code

- `atomic blocks`

- atomic blocks

- are available in three variations

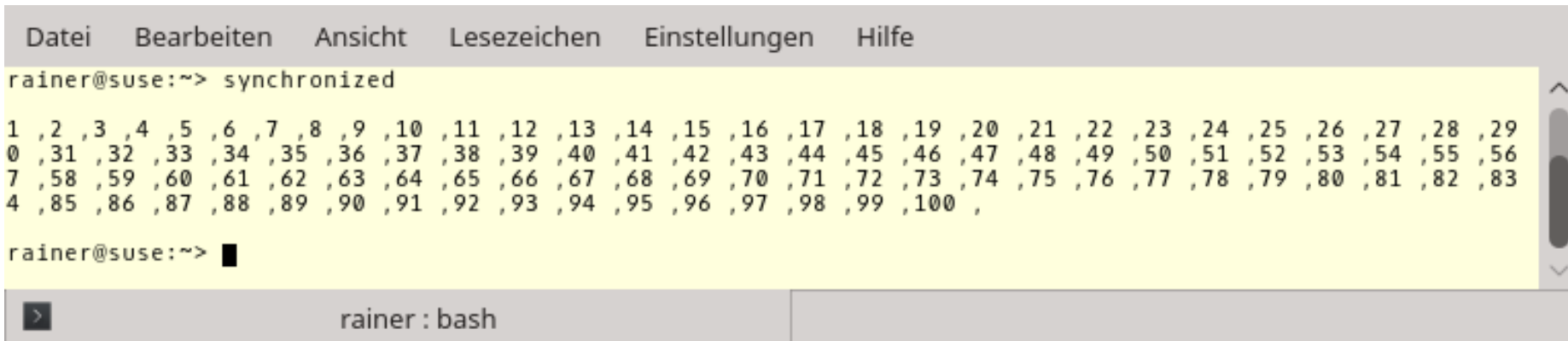
- ➔ can only execute `transaction-safe` code

# Transactional Memory

```
int i = 0;

void inc() {
    synchronized{
        cout << ++i << " ,";
    }
}

vector<thread> vecSyn(10);
for(auto& t: vecSyn)
    t = thread([]{ for(int n = 0; n < 10; ++n) inc(); });
```



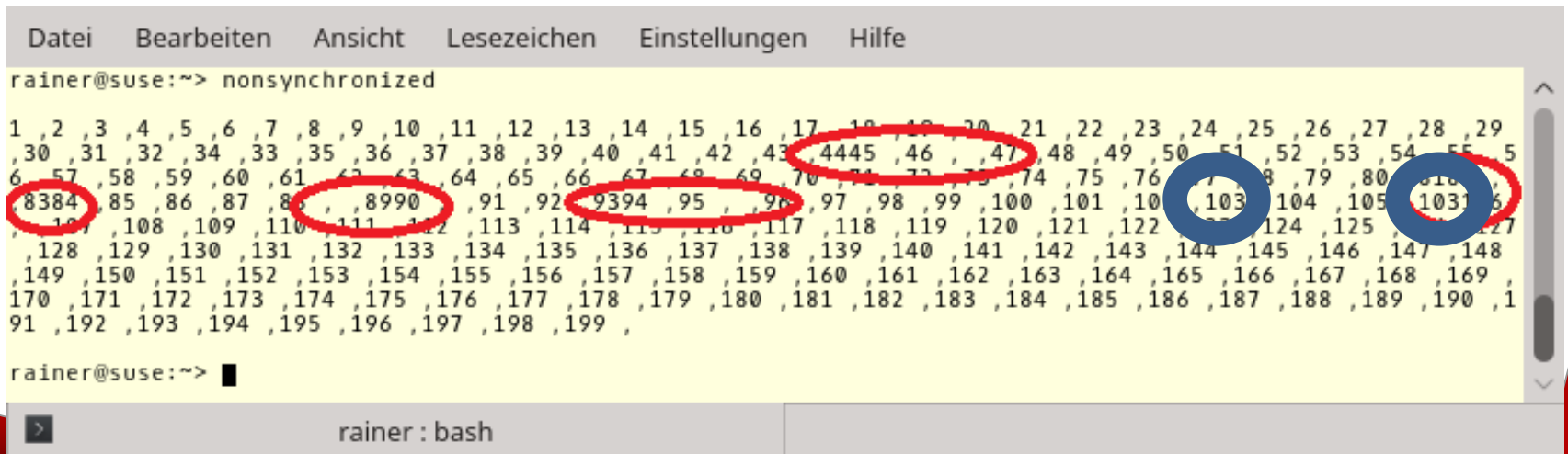
```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
rainer@suse:~> synchronized
1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14 ,15 ,16 ,17 ,18 ,19 ,20 ,21 ,22 ,23 ,24 ,25 ,26 ,27 ,28 ,29
0 ,31 ,32 ,33 ,34 ,35 ,36 ,37 ,38 ,39 ,40 ,41 ,42 ,43 ,44 ,45 ,46 ,47 ,48 ,49 ,50 ,51 ,52 ,53 ,54 ,55 ,56
7 ,58 ,59 ,60 ,61 ,62 ,63 ,64 ,65 ,66 ,67 ,68 ,69 ,70 ,71 ,72 ,73 ,74 ,75 ,76 ,77 ,78 ,79 ,80 ,81 ,82 ,83
4 ,85 ,86 ,87 ,88 ,89 ,90 ,91 ,92 ,93 ,94 ,95 ,96 ,97 ,98 ,99 ,100 ,
rainer@suse:~> █
```

> rainer : bash

# Transactional Memory

```
void inc() {  
    synchronized{  
        std::cout << ++i << " ,";  
        this_thread::sleep_for(1ns);  
    }  
}
```

```
vector<thread> vecSyn(10), vecUnsyn(10);  
for(auto& t: vecSyn)  
    t= thread[]{ for(int n = 0; n < 10; ++n) inc(); });  
for(auto& t: vecUnsyn)  
    t= thread[]{ for(int n = 0; n < 10; ++n) cout << ++i << " ,"; });
```



```
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe  
rainer@suse:~> nonsynchronized  
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,  
30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 4445, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,  
57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,  
8384, 85, 86, 87, 88, 89, 8990, 91, 92, 93, 9394, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107,  
108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127,  
128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148,  
149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169,  
170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191,  
192, 193, 194, 195, 196, 197, 198, 199,  
rainer@suse:~>   
rainer : bash
```

# Transactional Memory

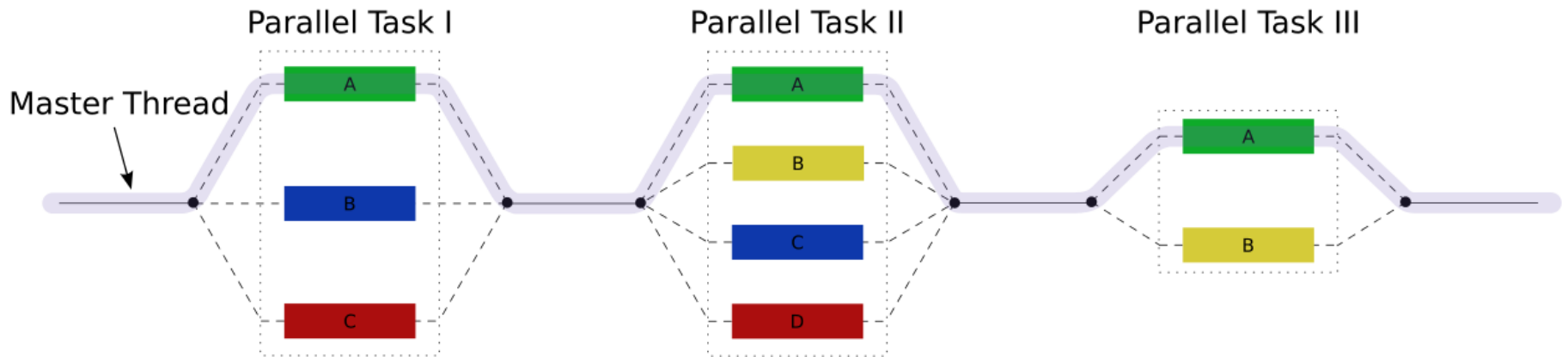
```
int i = 0;
void func() {
    atomic_noexcept{
        cout << ++i << " ,"; // non transaction-safe code
    }
}
```

A transaction can only perform `transaction-safe code`

➡ **compiler error**

# Task Blocks

Fork-join parallelism with task blocks.



# Task Blocks

```
template <typename Func>
int traverse(node& n, Func && f){
    int left = 0, right = 0;
    define_task_block(
        [&](task_block& tb) {
            if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
            if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
        }
    );
    return f(n) + left + right;
}
```

- **define\_task\_block**
  - tasks can be performed
  - tasks will be synchronised at the end of the task block
- **run**: starts a task

# Task Blocks

## define\_task\_block\_restore\_thread

```
...  
(1)  define_task_block([&](auto& tb)  
      tb.run([&]{} func(); });  
(2)  define_task_block_restore_thread([&](auto& tb){  
      tb.run([&]{} func2(); });  
(3)  define_task_block([&](auto& tb){  
      tb.run([&]{} func3(); }  
(3)  });  
      ...  
      ...  
(2)  });  
      ...  
      ...  
(1)  });  
      ...  
      ...
```

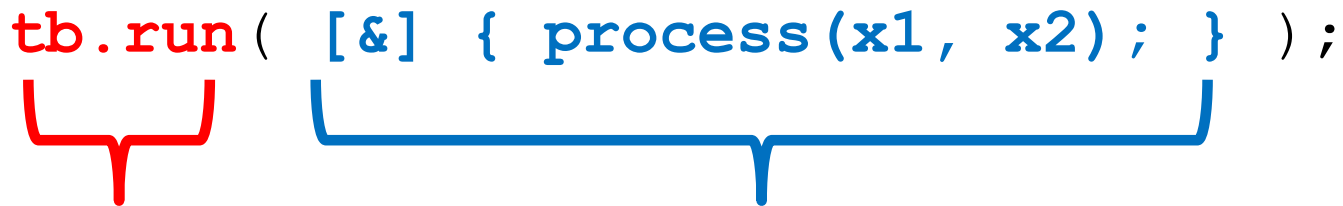
## wait

```
define_task_block([&](auto& tb){  
    tb.run([&]{} process(x1, x2); });  
    if (x2 == x3) tb.wait();  
    process(x3, x4);  
});
```

# Task Blocks

- The scheduler

```
tb.run ( [&] { process(x1, x2); } );
```



Parent

Child

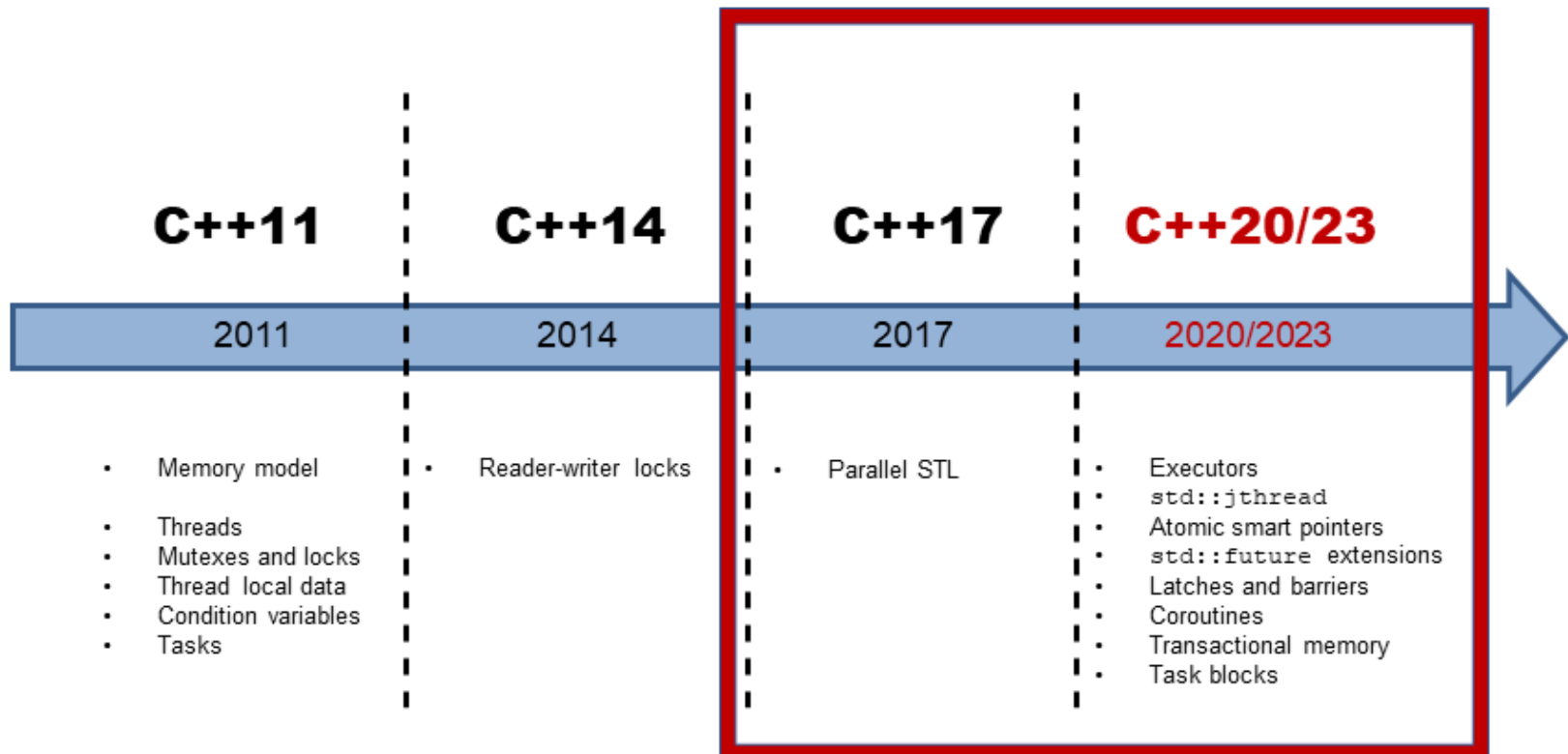
- **Child stealing:** the scheduler steals the job and executes it
- **Parent stealing:** the task block performs the child; the scheduler steals the parent

➡ Both strategies are possible in C++20

# Concurrency and Parallelism in C++

## Multithreading

## Concurrency and Parallelism



# Concurrency and Parallelism in C++



# Proposals and Drafts

- Unified Executors: [P0443R10](#) (2019), Dependent Executors [P1244R0](#) (2018)
- Stop Tokens and a Joining Thread [P0660R8](#) (2019)
- Atomic Smart Pointers: [N4162](#) (2014)
- `std::future` Extensions: [N4107](#) (2014) and [P070r1](#) (2017)
- Latches and Barriers: [P0666R0](#) (2017)
- Coroutines: [N4723](#) (2018)
- Transactional Memory: [N4265](#) (2014) => TM light (2019)
- Task Blocks: [N4411](#) (2015)
  
- Working Draft, Standard for Programming Language C++ ([N4800](#))

# Blogs

[www.grimm-jaud.de](http://www.grimm-jaud.de) [De]

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

Rainer Grimm

Training, Coaching, and  
Technology Consulting

[www.ModernesCpp.de](http://www.ModernesCpp.de)