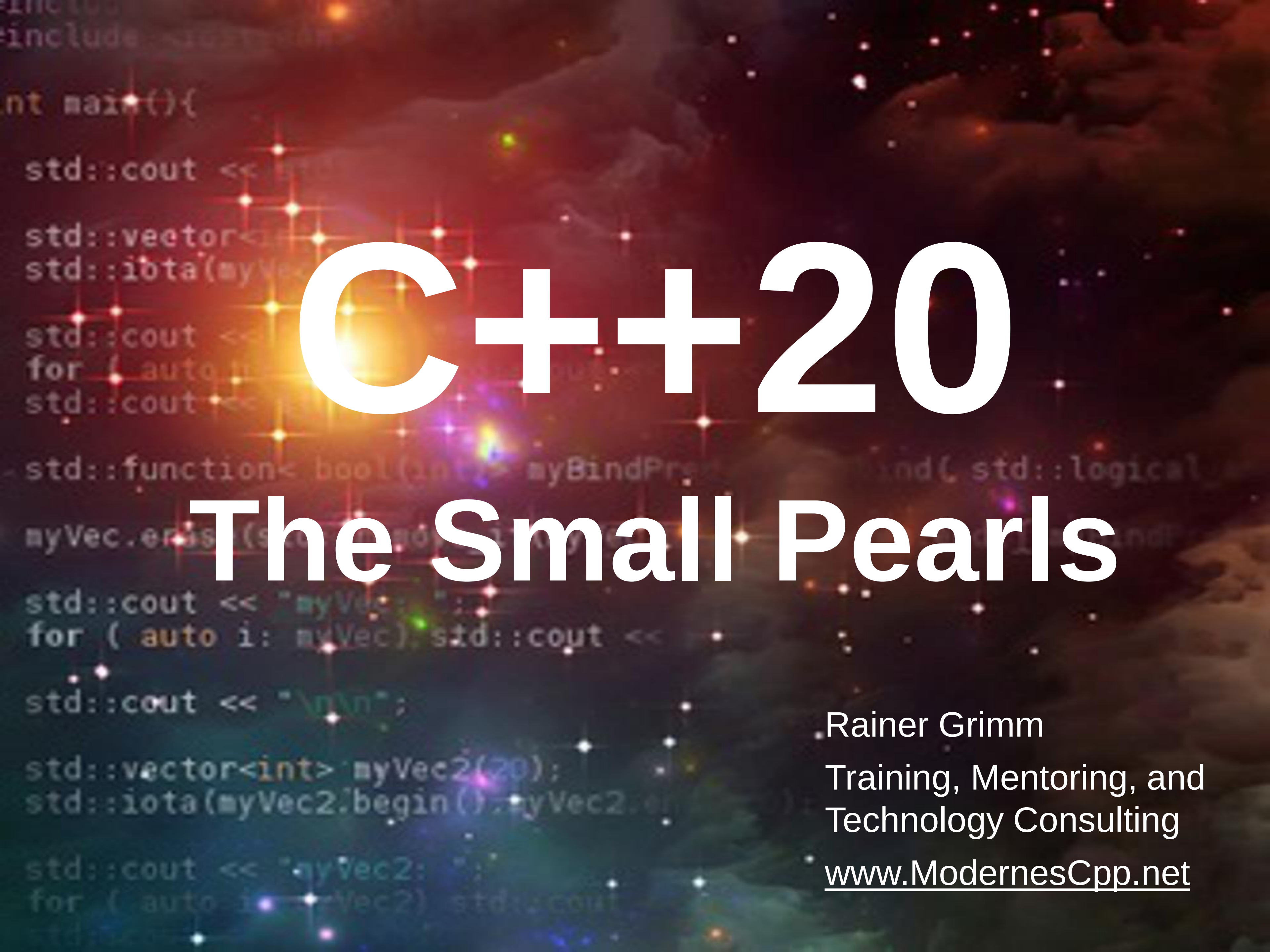


C++20

The Small Pearls

Rainer Grimm

Training, Mentoring, and
Technology Consulting

www.ModernesCpp.net

C++20

2020

The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

Core Language

- Three-way comparison operator
- Designated initialization
- `consteval` and `constinit`
- Template improvements
- Lambda improvements

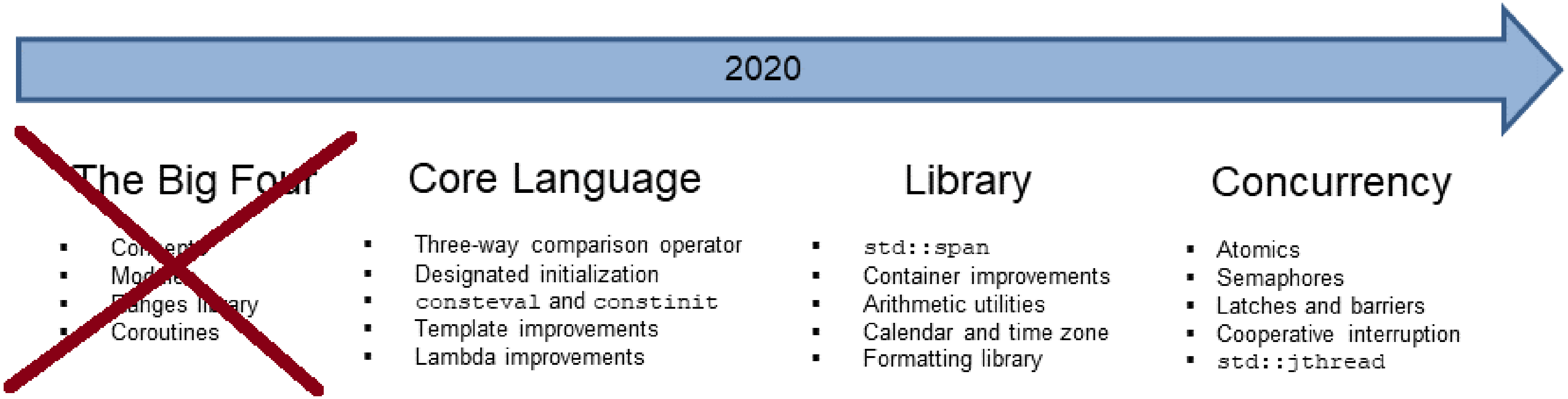
Library

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

Concurrency

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

C++20 – The Big Four



2020

~~The Big Four~~

- ~~Concepts~~
- ~~Modules~~
- ~~Ranges library~~
- ~~Coroutines~~

Core Language

- Three-way comparison operator
- Designated initialization
- `consteval` and `constinit`
- Template improvements
- Lambda improvements

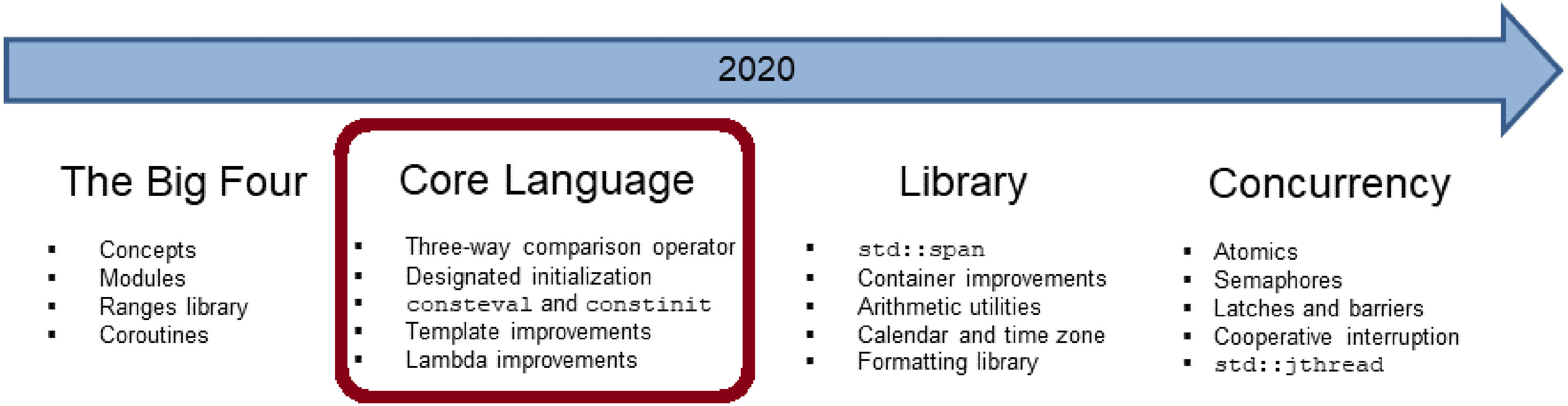
Library

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

Concurrency

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

C++20 - Core Language



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`

Three-way Comparison Operator

The three-way comparison operator `<=>` determines for two values A and B, whether A < B, A == B or A > B applies.

- The three-way comparison operator
  - is also called spaceship operator.
  - can be implemented or defaulted with `= default`.

- The comparison operator created by the compiler
  - needs the header file `<compare>`.
  - is implicit `constexpr` and `noexcept`.
  - compares lexicographically except the `==` and `!=` operator.
    - All base classes from left to right
    - Non-static members in their declaration order

Three-way Comparison Operator

User defined

```
struct MyInt {
  int value;
  explicit MyInt(int val): value{val} {}
  auto operator<=>(const MyInt& rhs) const {         // strong ord.
      return value <=> rhs.value;
  }
};
```

Compiler generated

```
struct MyDouble {
  double value;
  explicit MyDouble(double val): value{val} {}
  auto operator<=>(const MyDouble&) const = default; // partial ord.
};
```

Three-way Comparison Operator

- Special features

  - The compiler generates comparison expressions from the three-way comparison order:

    ```
    a < b ➡ (a <=> b) < 0
    ```

  - The three-way comparison operator is symmetric.

    ```
    a < b ➡ (a <=> b) < 0 ➡ 0 < (b <=> a)
    ```

  - If the data type already has comparison operators, they have higher priority than the three-way comparison operator.

threeWayComparisonWithInt.cpp

Designated Initialization

Designated initializers are an extension of aggregate initialization.

- Aggregate
  - Array
  - Class type (`class, struct, union`)
    - `public` members or base classes
    - No user-defined constructors
    - No virtual members or base classes
- Aggregate Initialization
  - Can be initialized directly with an initialization list.
  - The order of the arguments must match the declaration order of the members.

Designated Initialization

```
Point {
    int x;
    int y;
};
```

Designated Initializer

- Allows to call the non-static members directly by name using an initializer list.
    - `Point p = {.x = 1, .y = 2};`
- Members can also have an in-class default value.
- If the initializer is missing, the default value is used (exception `union`).

- Narrowing conversion is detected ➡ ERROR

designatedInitializerDefaults.cpp

`consteval`

`consteval` generates an *immediate* function.

- Every call of an *immediate* function generates a constant expression that is executed at compile time.

`consteval`

- Cannot be applied to destructors or functions that allocate or deallocate.
- Has the same requirements such as a `constexpr` function.
- Implies that the function is `inline`.

```
consteval int sqr(int n) {
    return n * n;
}
constexpr int r = sqr(100);  // OK

int x = 100;
int r2 = sqr(x);             // Error
```

`constinit`

`constinit` guarantees that a variable with static storage duration is initialized at compile time.

- Global objects, or objects declared with `static` or `extern`, have static storage duration.
- Objects with a static storage duration are allocated at the program start and deallocated at its end.

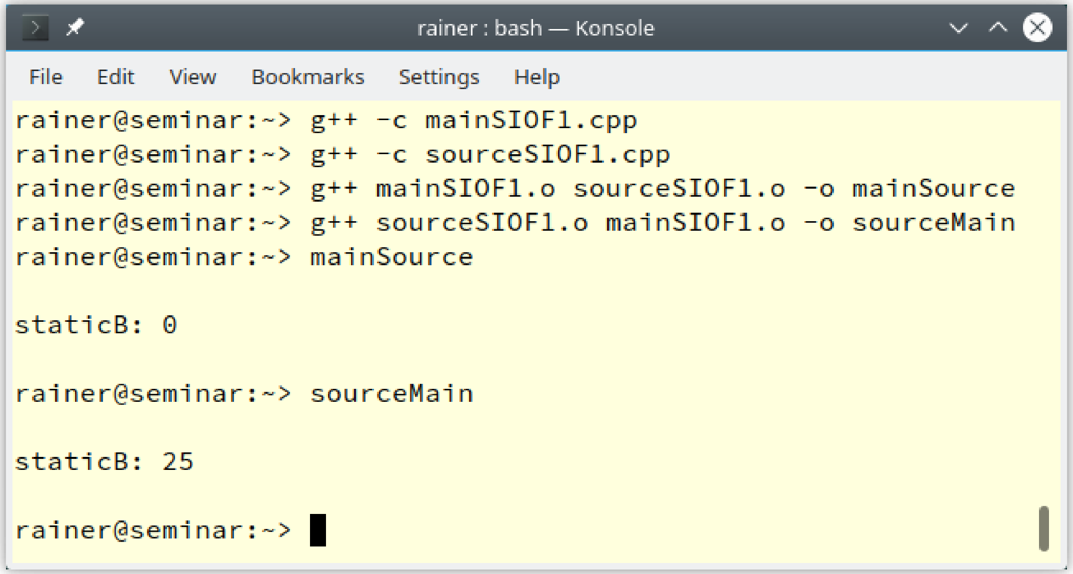
`constinit`

- Avoids the static initialization order fiasco.
- Variables are not constant.

constinit

```
// sourceSIOF1.cpp
int square(int n) {
  return n * n;
}
auto staticA = square(5);
```

```
// mainSOIF1.cpp
#include <iostream>

extern int staticA;
auto staticB = staticA;

int main() {
  std::cout << "staticB: " << staticB;
}
```



```
rainer@seminar:~> g++ -c mainSIOF1.cpp
rainer@seminar:~> g++ -c sourceSIOF1.cpp
rainer@seminar:~> g++ mainSIOF1.o sourceSIOF1.o -o mainSource
rainer@seminar:~> g++ sourceSIOF1.o mainSIOF1.o -o sourceMain
rainer@seminar:~> mainSource

staticB: 0

rainer@seminar:~> sourceMain

staticB: 25

rainer@seminar:~>
```

Template and Lambda Improvements

- New non-type template-parameters
  - Floating-point numbers
  - Classes with `constexpr` constructor

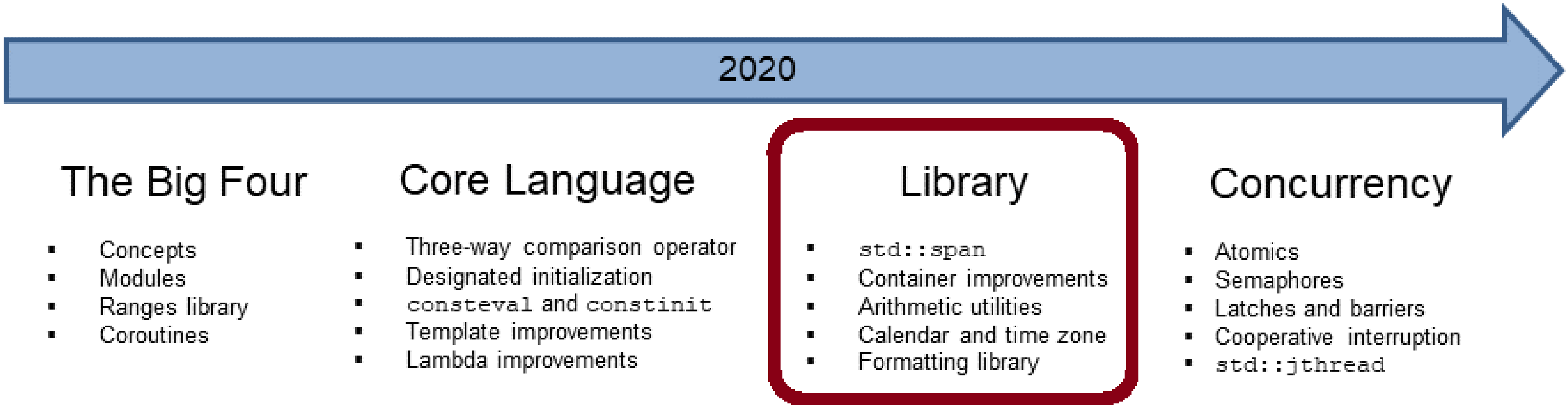
- Template lambdas allow defining a lambda expression that can only be used for certain types.

```
auto foo = []<typename T>(const std::vector<T>& vec) {
        // do vector specific stuff
    };
```

A concept can be used instead of a type parameter `T`.

templateLambda.cpp

C++20 - Library



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`

`std::span`

`std:span` stands for an object that refers to a continuous sequence of objects.

- `std::span`
  - is never an owner.
  - The referenced area can be an array, a pointer with a length, or a `std::vector.`
  - A typical implementation has a pointer to the first element and its length.
  - Allows the partially access to the continuous sequence of elements.

⇨ A `std::span` knows its length.

printSpan.cpp

std::span

Modifying a span also modifies the referenced objects.

```
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
printMe(vec);       // displays size and elements
std::span span1(vec);
std::span span2{span1.subspan(1, span1.size() - 2)};
std::transform(span2.begin(), span2.end(),
               span2.begin(), [](int i){ return i * i; });
printMe(vec);
printMe(span1);
```

spanTransform.cpp

Container Improvements

`std::string` and `std::vector` can be created and modified at compile time.

- The constructors of `std::string`, and `std::vector` constructors and member functions are `constexpr`.
- The algorithms of the Standard Template Library are declared `constexpr`.

If a function is declared as `constexpr`, it has the potential to run at compile time.

`constexprVector.cpp`

Container Improvements

`std::erase` and `std::erase_if` enable the uniform deletion of the elements of a container.

- `std::erase(container, value):`
  - Removes all elements with the `value` from the `container.`
- `std::erase_if(container, predicate):`
  - Removes all elements from the `container` that fulfil the `predicate.`

! Both algorithms operate directly on the container.

`eraseUpper.cpp`

Container Improvements

`std::string str`

`str.starts_with(prefix):`

- Checks if the string `str` starts with the given `prefix`.

`str.ends_with(suffix):`

- Checks if the string `str` ends with the given `suffix`.

Arithmetic Utilities

The comparison of signed and unsigned integers often does not yield the expected result.

- The `std::cmp_*`-functions perform a secure comparison.

Compare Function	Meaning
`std::cmp_equal`	`==`
`std::cmp_not_equal`	`!=`
`std::cmp_less`	`<`
`std::cmp_less_equal`	`<=`
`std::cmp_greater`	`>`
`std::cmp_greater_equal`	`>=`

➡ It causes a compile time error if an argument is not an integer.

[safeComparison.cpp]()

Arithmetic Utilities

C++20 supports important mathematical constants.

- Need the header file `<numbers>`
- Are defined in the namespace `std::numbers`
- The constants have the data type `double`.

Constant	Meaning
`e`	e
`log2e`	log_2e
`log10e`	$log_{10}e$
`pi`	π
`inv_pi`	$\frac{1}{\pi}$
`inv_sqrtpi`	$\frac{1}{\sqrt{\pi}}$

Constant	Meaning
`ln2`	$ln2$
`ln10`	$ln10$
`sqrt2`	$\sqrt{2}$
`sqrt3`	$\sqrt{3}$
`inv_sqrt3`	$\frac{1}{\sqrt{3}}$
`egamma`	Euler-Mascheroni constant
`phi`	ϕ $(\frac{1+\sqrt{5}}{2})$

Calendar and Time Zones

The chrono library is extended by additional clocks, time of day, a calendar, and time zones.

- **New Clocks**
  - `std::chrono::utc_clock`
  - `std::chrono::tai_clock`
  - `std::chrono::gsp_clock`
  - `std::chrono::file_clock`
  - `std::chrono::local_clock`
- **Time of Day:**
  - Time since midnight in the format hours:minutes:seconds.

Calendar and Time Zones

- **Calendar:**
  - Data types representing a year, a month, a weekday, and the n-th day of the week.
  - Data types can be combined to more complex data types.
  - The "`/`" operator allows easy handling of time points.
  - C++ has two new literals: `d` for a day and `y` for a year.
- **Time zones:**
  - Display dates in different time zones.

`timeOfDay.cpp`
`cuteSyntax.cpp`
`localTime.cpp`
`onlineClass.cpp`

Formatting Library

The formatting library offers a secure and expandable alternative to the `printf` family and extends the I/O streams.

The formatting library requires header file `<format>`.

The format specifications follow the Python syntax.

- The format specification allows to
  - Specify fill letters and text alignment.
  - Set the sign for numbers.
  - Specify the width and precision of numbers.
  - Specify the data type.

Formatting Library

- `std::format`
  - Returns the formatted string.

- `std::format_to`
  - Writes the formatted output using an output iterator.

- `std::format_to_n`
  - Writes a maximum of `n` characters of the formatted output using an output iterator.

⇨ All three functions follow the same syntax.

Formatting Library

Syntax: `std::format(FormatString, Arguments)`

```
std::format("{1} {0}!", "world", "Hello");
```

- The `FormatString` consists of
  - Characters: are not changed (exception **{** and **}**)
  - Escape sequences: **{{** and **}}** become **{** and **}**
  - Replacement fields:
    - Introductory character: {
    - Argument-ID: optional, followed by a format specifier
    - Colon: optional; introduces the format specifier
    - End character: }

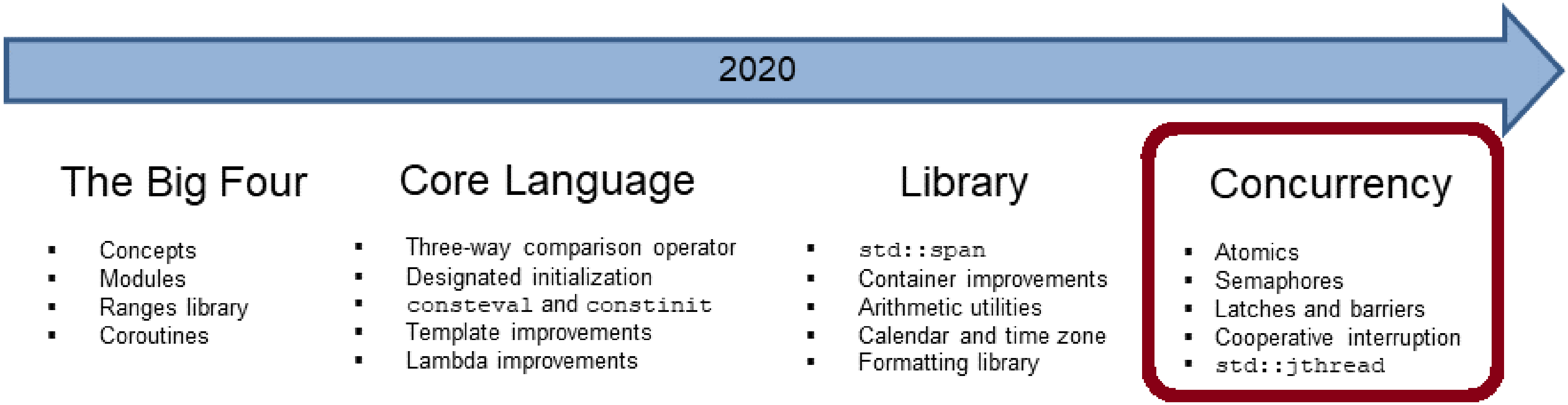
Formatting Library

The format specifier `std::formatter` provides formatting rules for data types.

- Elementary data types and `std::string`:
  - Standard format specification based on Python’s format specification
- Chrono data types:
  - `chrono` format specification
- Further data types:
  - User-defined format specification

`formatArgumentID.cpp`
`formatVector.cpp`

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`

Atomics

`std::atomic` offers specializations for `float`, `double` and `long double.`

- `std::atomic` and `std::atomic_flag`
  - Allow 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

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>)`

threadSynchronisationSemaphore.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();`

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.

[`workers.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.

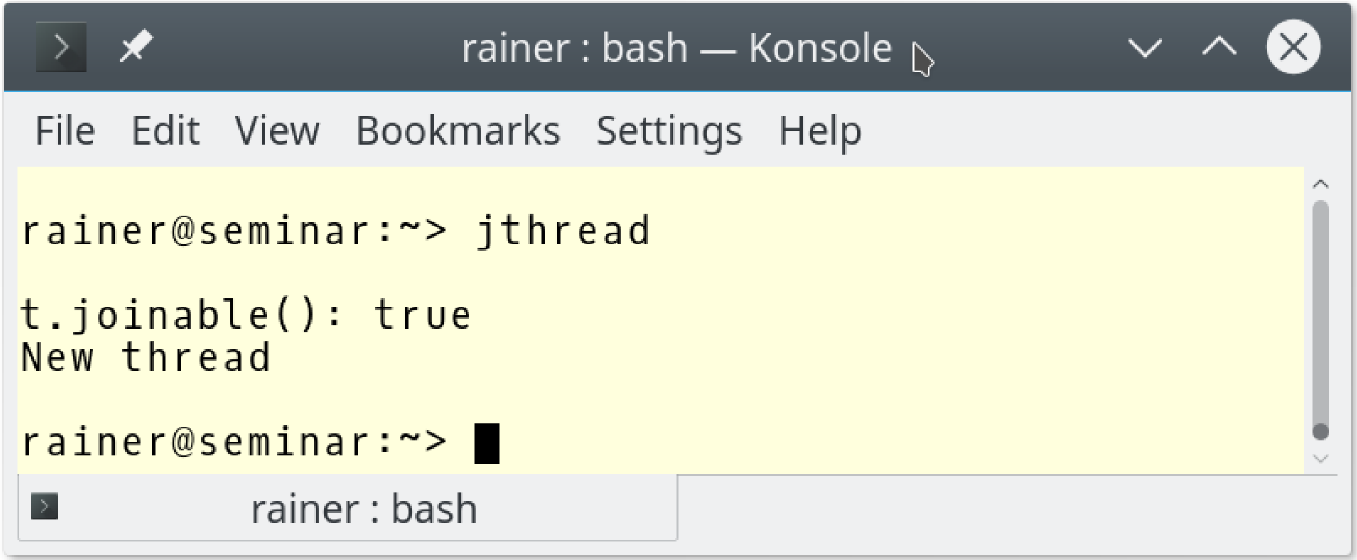
```
std::stop_source stopSource;
std::stop_token stopToken = stopSource.get_token();

void function(std::stop_token stopToken){
    if (stopToken.stop_requested()) return;
}

std::thread thr = std::thread(function, stopToken);
stopSource.request_stop();
```

[stopRequested.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:~>
```

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

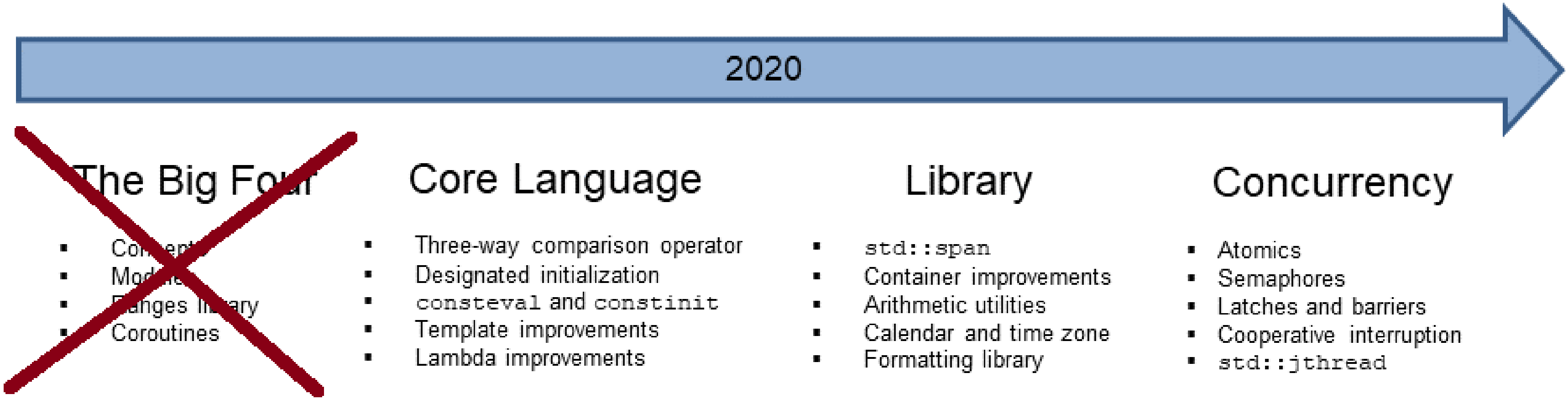
- Permanent variable `synced_out`

```
{
    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!"
                            << std::endl;
```

C++20 – The Big Four



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

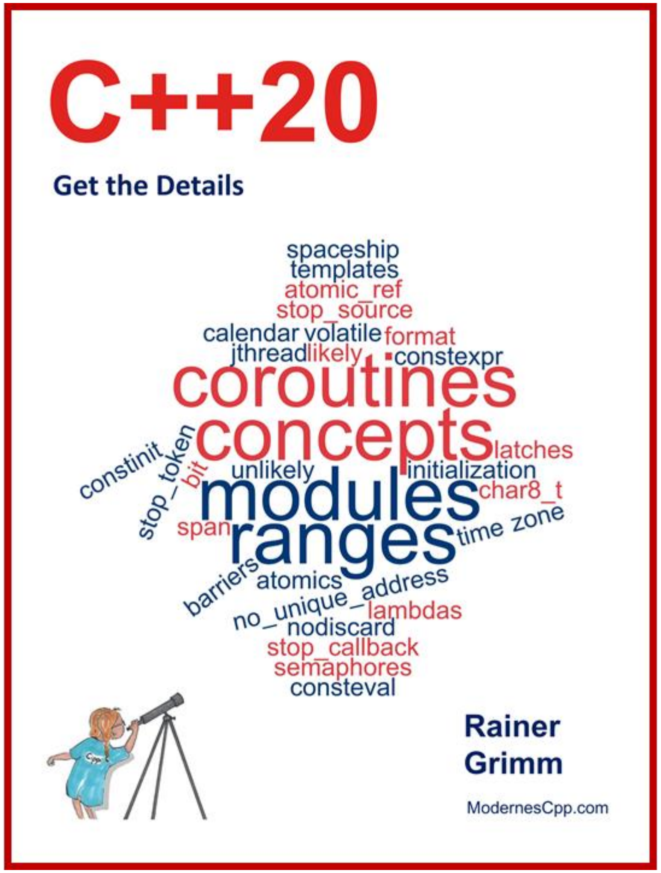
Concurrency

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

C++20

- Modernes C++ Blog

- C++20: Get the Details



C++20

Get the Details

Rainer
Grimm

ModernesCpp.com

www.ModernesCpp.com

Rainer Grimm

Training, Mentoring, and Technology Consulting

www.ModernesCpp.net