

```
#include <vector>
```

```
int main(){
```

```
    std::cout << "myVec: ";
```

```
    std::vector<int> myVec(10);  
    std::iota(myVec.begin(), myVec.end(), 1);
```

```
    std::cout << " ";  
    for ( auto i: myVec ) std::cout << i << " ";  
    std::cout << "\n\n";
```

```
    std::function< bool(int)> myBindPrint = bind( std::cout, std::endl );
```

```
    myVec.erase(myVec.begin(), myVec.end());
```

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

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

```
    std::vector<int> myVec2(20);  
    std::iota(myVec2.begin(), myVec2.end(), 1);
```

```
    std::cout << "myVec2: ";  
    for ( auto i: myVec2 ) std::cout << i << " ";  
    std::cout << "\n\n";
```

C++20

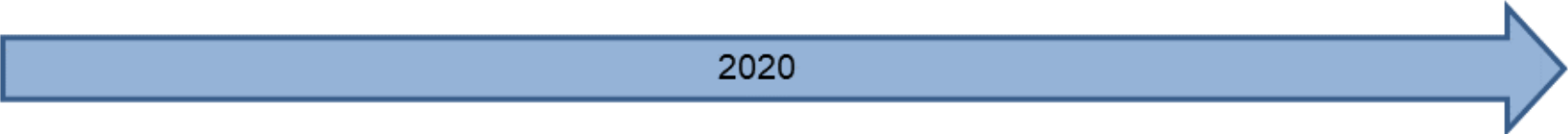
The Hidden Pearls

Rainer Grimm

Training, Coaching, 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
- `constexpr` and `constexpr`
- Template improvements
- Lambda improvements

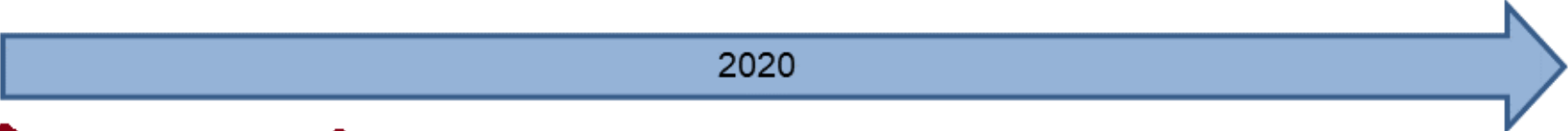
Library

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

Concurrency

- `std::atomic`
- 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
- `constexpr` 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

2020



The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

Core Language

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

Library

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

Concurrency

- `std::atomic`
- 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

- Special features
 - The compiler generates comparison expressions from the three-way comparison order:
 $a < b \Rightarrow (a <=> b) < 0$
 - The three-way comparison operator is symmetric.
 $a < b \Rightarrow (a <=> b) < 0 \Rightarrow 0 < (b <=> a)$
 - If the data type already has comparison operators, they have higher priority than the three-way comparison operator.

Designated Initialization


Designated initializers are an extension of aggregate initialization.

- **Aggregate**
 - Array
 - **Class type** (`class`, `struct`, `union`)
 - `public` members
 - No user-defined constructors
- **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

constexpr

`constexpr` generates an *immediate* function.

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

`constexpr`

- 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`.

```
constexpr 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.

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.

C++20 - Library

2020



The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

Core Language

- Three-way comparison operator
- Designated initialization
- `constexpr` and `constexpr`
- 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](#)

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.

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.

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
<code>std::cmp_equal</code>	<code>==</code>
<code>std::cmp_not_equal</code>	<code>!=</code>
<code>std::cmp_less</code>	<code><</code>
<code>std::cmp_less_equal</code>	<code><=</code>
<code>std::cmp_greater</code>	<code>></code>
<code>std::cmp_greater_equal</code>	<code>>=</code>

 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
<code>e</code>	e
<code>log2e</code>	$\log_2 e$
<code>log10e</code>	$\log_{10} e$
<code>pi</code>	π
<code>inv_pi</code>	$\frac{1}{\pi}$
<code>inv_sqrtpi</code>	$\frac{1}{\sqrt{\pi}}$

Constant	Meaning
<code>ln2</code>	$\ln 2$
<code>ln10</code>	$\ln 10$
<code>sqrt2</code>	$\sqrt{2}$
<code>sqrt3</code>	$\sqrt{3}$
<code>inv_sqrt3</code>	$\frac{1}{\sqrt{3}}$
<code>egamma</code>	Euler-Mascheroni constant
<code>phi</code>	$\phi \left(\frac{1+\sqrt{5}}{2} \right)$

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::gps_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: `}`

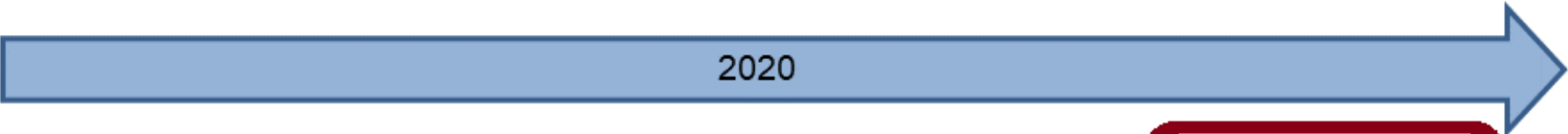
Formatting Library

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
- `constexpr` and `constexpr`
- 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.

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`

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>`)

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
<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

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

Member Function	Description
<code>bar.arrive(upd)</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>wait(arrive())</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 callabe is executed by an arbitrary thread.

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 token`)

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

Cooperative Interruption

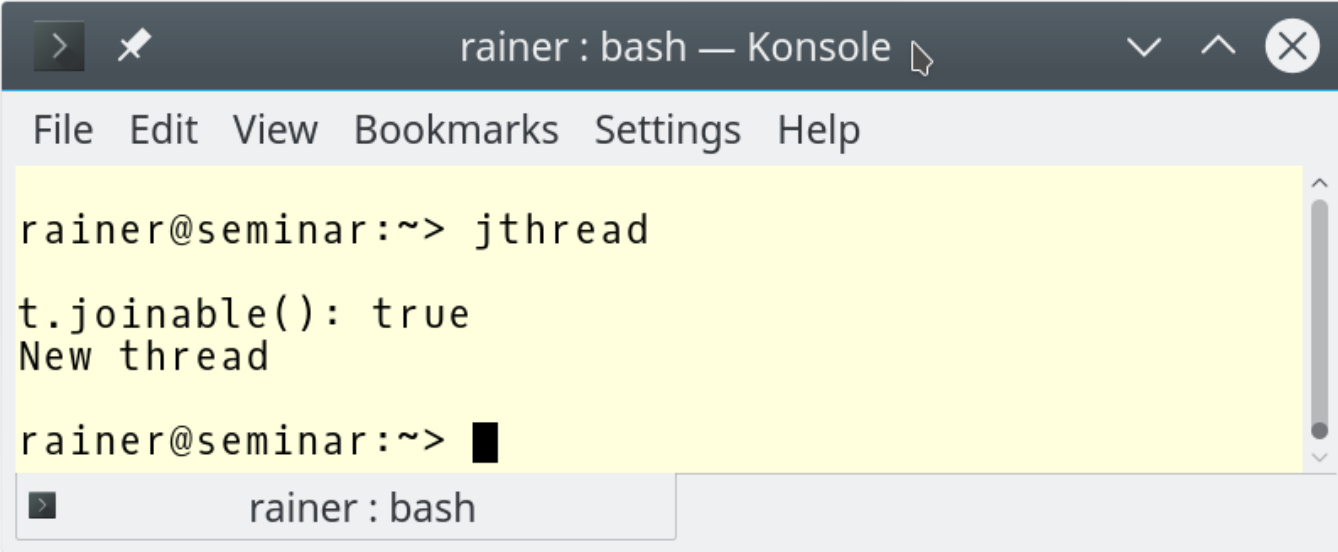
Sender (`std::stop_source`)

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.

std::jthread

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

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

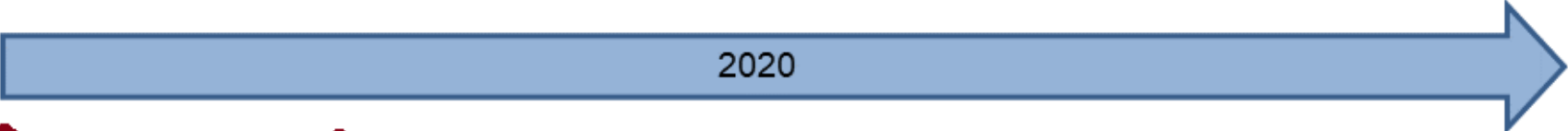


The screenshot shows a terminal window titled "rainer : bash — Konsole". The terminal output is as follows:

```
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. At the bottom, there is a tab labeled "rainer : bash".

C++20 – The Big Four



2020

~~The Big Four~~

- Concepts
- Modules
- Ranges library
- Coroutines

Core Language

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

Library

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

Concurrency

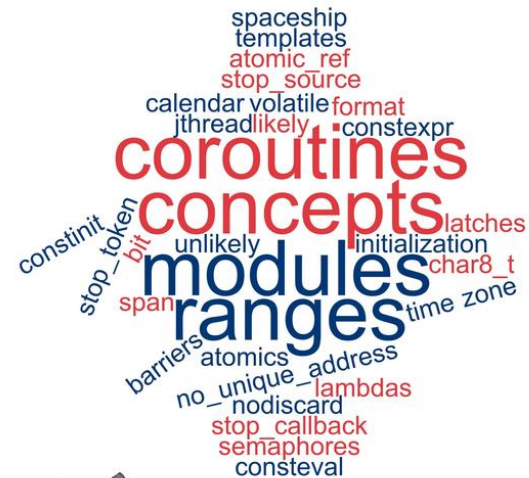
- Atomics
- Semaphores
- Latches and barriers
- Cooperative interruption
- `std::jthread`

C++20

- [Modernes C++ Blog](#)
- [C++20: Get the Details \(50 % off until Sunday\)](#)

C++20

Get the Details



**Rainer
Grimm**

ModernesCpp.com



```
#include <vector>
#include <string>
#include <functional>
using namespace std;

int main(){

    std::cout << "myVec: ";
    std::vector<int> myVec(10);
    std::iota(myVec.begin(), myVec.end(), 1);
    std::cout << " ";
    for ( auto i: myVec) std::cout << i << " ";
    std::cout << "\n";

    std::function< bool(int)> myBindPred = bind( std::logical_not(),
    myVec.erase(std::remove_if(myVec.begin(), myVec.end(), myBindPred),
    myVec.end());

    std::cout << "myVec: ";
    for ( auto i: myVec) std::cout << i << " ";
    std::cout << "\n";

    std::vector<int> myVec2(20);
    std::iota(myVec2.begin(), myVec2.end(), 1);
    std::cout << "myVec2: ";
    for ( auto i: myVec2) std::cout << i << " ";
    std::cout << "\n";
}
```

www.ModernesCpp.com

Rainer Grimm
Training, Coaching, and
Technology Consulting
www.ModernesCpp.net