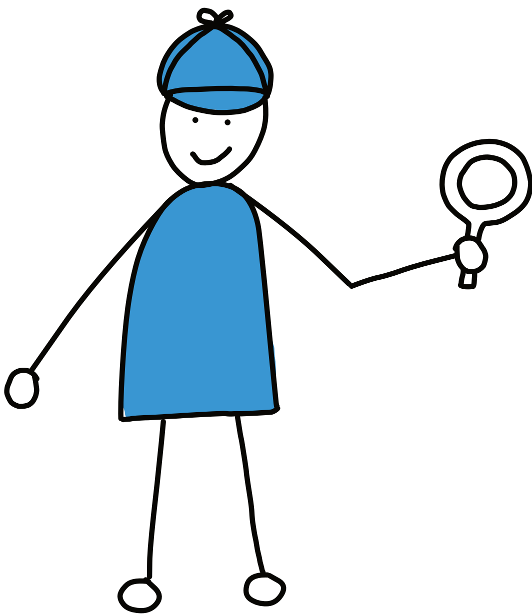
C++ Coroutines from scratch

Presentation Material



CppIndia Con, Online, 2023-08-05

Andreas Fertig

Write unique code!

© 2023 Andreas Fertig
AndreasFertig.com
All rights reserved

All programs, procedures and electronic circuits contained in this book have been created to the best of our knowledge and belief and have been tested with care. Nevertheless, errors cannot be completely ruled out. For this reason, the program material contained in this book is not associated with any obligation or guarantee of any kind. The author therefore assumes no responsibility and will not accept any liability, consequential or otherwise, arising in any way from the use of this program material or parts thereof.

Version: v1.0

The work including all its parts is protected by copyright. Any use beyond the limits of copyright law requires the prior consent of the author. This applies in particular to duplication, processing, translation and storage and processing in electronic systems.

The reproduction of common names, trade names, product designations, etc. in this work does not justify the assumption that such names are to be regarded as free in the sense of trademark and brand protection legislation and can therefore be used by anyone, even without special identification.

Planning, typesetting and cover design: Andreas Fertig
Cover art and illustrations: Franziska Panter https://franziskapanter.com
Production and publishing: Andreas Fertig

Style and conventions

The following shows the execution of a program. I used the Linux way here and skipped supplying the desired output name, resulting in `a.out` as the program name.

```
$ ./a.out
Hello, C++!
```

- `<string>` stands for a header file with the name `string`
- `[[xyz]]` marks a C++ attribute with the name `xyz`.

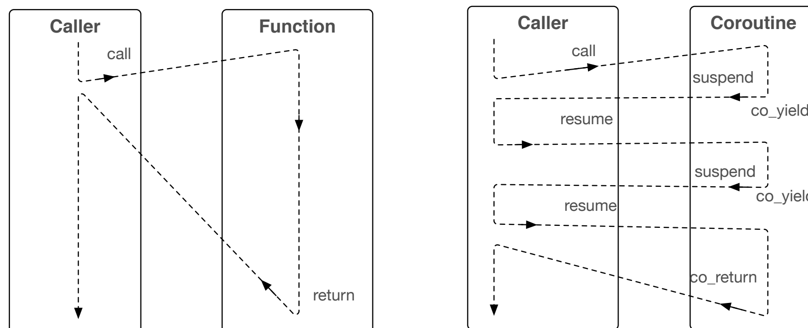
© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed

Function vs. Coroutine comparison



© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

What are Coroutines?

- The term coroutine has been well-established in computer science since it was first coined in 1958 by Melvin Conway [1].
- They come in two different forms:
  - Stack*full*
  - Stack*less* (which is what we have in C++)
- Stackless means that the data of a coroutine, the coroutine frame, is stored on the heap.
- We are talking about cooperative multitasking when using coroutines.
- Coroutines can simplify your code!
  - We can replace some function pointers (callbacks) with coroutines.
  - Parsers are much more readable with coroutines.
  - A lot of state maintenance code is no longer required as the coroutine does the bookkeeping.

Interacting with a coroutine

- Coroutines can be paused and resumed.
- `co_yield` or `co_await` pause a coroutine.
- `co_return` ends a coroutine.

Keyword	Action	State
`co_yield`	Output	Suspended
`co_return`	Output	Ended
`co_await`	Input	Suspended

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Elements of a Coroutine

- In C++, a coroutine consists of:
  - A wrapper type. This is the return type of the coroutine function's prototype.
    - With this type can control the coroutine from the outside. For example, resuming the coroutine or getting data into or from the coroutine by storing a handle to the coroutine in the wrapper type.
  - The compiler looks for a *type* with the exact name `promise_type` inside the return type of the coroutine (the wrapper type). This is the control from the inside.
    - This type can be a type alias, or
    - a **`typedef`**,
    - or you can declare the type directly inside the coroutine wrapper type.
  - An awaitable type that comes into play once we use **`co_await`**.
  - We also often use another part, an iterator.
- A coroutine in C++ is an finite state machine (FSM) that can be controlled and customized by the `promise_type`.
- The actual coroutine function which uses **`co_yield`**, **`co_await`**, or **`co_return`** for communication with the world outside.

Disclaimer

Please note, I tried to keep the code you will see as simple as possible. Focusing on coroutines. In production code, I work more with **`public`** and **`private`** as well as potential getters and setters. Additionally, I use way more generic code in production code to keep repetitions low.
My goal is to help you understand coroutines. I'm confident that you can improve the code you will see with the usual C++ best practices.
I also never declare more than one variable per line... slide code is the only exception.

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Coroutine chat

```
1  Chat Fun()  (A) Wrapper type Chat containing the promise type
2  {
3    co_yield "Hello!\n"s;  (B) Calls promise_type.yield_value
4
5    std::cout << co_await std::string{};  (C) Calls promise_type.await_transform
6
7    co_return "Here!\n"s;  (D) Calls promise_type.return_value
8  }
9
10 void Use()
11 {
12   Chat chat = Fun();  (E) Creation of the coroutine
13
14   std::cout << chat.listen();  (F) Trigger the machine
15
16   chat.answer("Where are you?\n"s);  (G) Send data into the coroutine
17
18   std::cout << chat.listen();  (H) Wait for more data from the coroutine
19 }
```


Coroutine chat

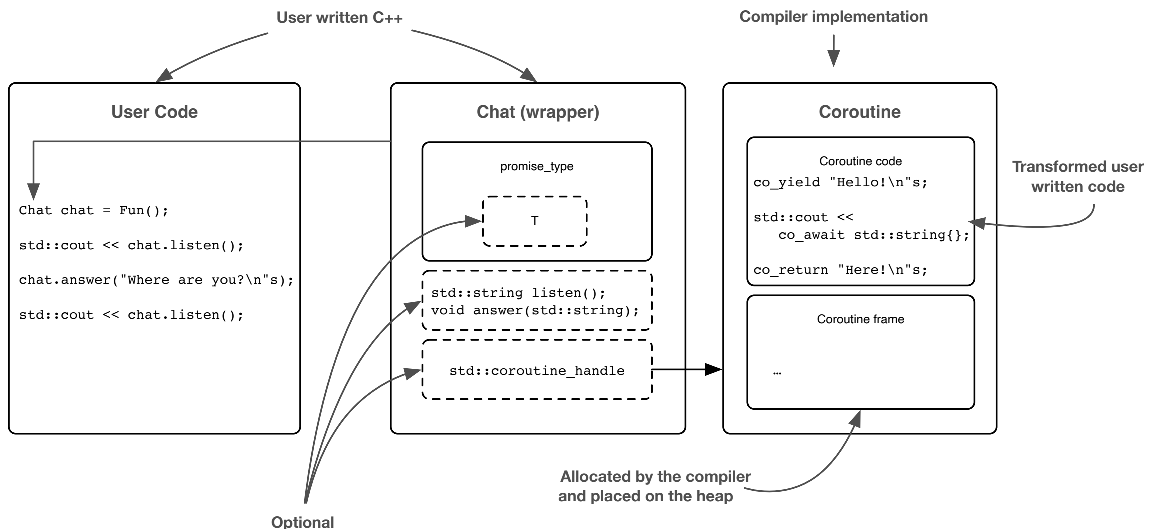
```
1  struct promise_type {
2    std::string _msgOut{}, _msgIn{};  (A) Storing a value from or for the coroutine
3
4    void                 unhandled_exception() noexcept {}          (B) What to do in case of an exception
5    Chat                 get_return_object() { return Chat{this}; }  (C) Coroutine creation
6    std::suspend_always initial_suspend() noexcept { return {}; }   (D) Startup
7    std::suspend_always yield_value(std::string msg) noexcept       (F) Value from co_yield
8    {
9      _msgOut = std::move(msg);
10     return {};
11   }
12
13   auto await_transform(std::string) noexcept  (G) Value from co_await
14   {
15     struct awaiter {  (H) Customized version instead of using suspend_always or suspend_never
16       promise_type&  pt;
17       constexpr bool await_ready() const noexcept { return true; }
18       std::string    await_resume() const noexcept { return std::move(pt._msgIn); }
19       void           await_suspend(std::coroutine_handle<>) const noexcept {}
20     };
21
22     return awaiter{*this};
23   }
24
25   void return_value(std::string msg) noexcept { _msgOut = std::move(msg); }  (I) Value from co_return
26   std::suspend_always final_suspend() noexcept { return {}; }                (E) Ending
27 };
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Coroutine chat

```cpp
1 struct Chat {
2 #include "promise-type.h"  // Don't do that at work!
3
4   using Handle = std::coroutine_handle<promise_type>;   A Shortcut for the handle type
5   Handle mCoroHdl{};                                    B
6
7   explicit Chat(promise_type* p) : mCoroHdl{Handle::from_promise(*p)} {}  C Get the handle form the promise
8   Chat(Chat&& rhs) noexcept : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)} {}  D Move only!
9
10  ~Chat() noexcept  E Care taking, destroying the handle if needed
11  {
12    if(mCoroHdl) { mCoroHdl.destroy(); }
13  }
14
15  std::string listen()  F Activate the coroutine and wait for data.
16  {
17    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
18    return std::move(mCoroHdl.promise()._msgOut);
19  }
20
21  void answer(std::string msg)  G Send data to the coroutine and activate it.
22  {
23    mCoroHdl.promise()._msgIn = std::move(msg);
24    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
25  }
26 };
```


Coroutine customization points



© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

A few definitions

- Task: A coroutine that does a job without returning a value.
- Generator: A coroutine that does a job and returns a value (either by **co_return** or **co_yield**).

Helper types for Coroutines

- For yield_value, initial_suspend, final_suspend, as well as **co_await** / await_transform, we have two helper types in the Standard Template Library (STL):
  - std::suspend_always: The method await_ready always returns **false**, indicating that an await expression always suspends as it waits for its value.
  - std::suspend_never: The method await_ready always returns **true**, indicating that an await expression never suspends.

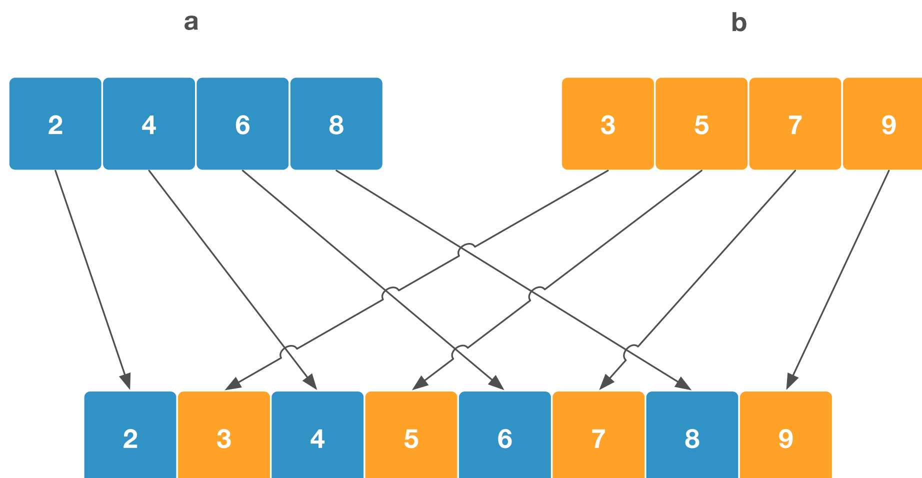
```
1 struct suspend_always {
2   constexpr bool await_ready() const noexcept
3   {
4     return  false ;
5   }
6
7   constexpr void
8   await_suspend(std::coroutine_handle<>) const noexcept
9   {}
10
11   constexpr void await_resume() const noexcept {}
12 };
```

```
1 struct suspend_never {
2   constexpr bool await_ready() const noexcept
3   {
4     return  true ;
5   }
6
7   constexpr void
8   await_suspend(std::coroutine_handle<>) const noexcept
9   {}
10
11   constexpr void await_resume() const noexcept {}
12 };
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Another task for a coroutine: Interleave two `std::vector` objects.

Interleaving two `std::vector`s



© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Interleaving two std::vector s

- The interleave coroutine function.

```
1 Generator interleaved(std::vector<int> a, std::vector<int> b)
2 {
3   auto lamb = [](std::vector<int>& v) -> Generator {
4     for(const auto& e : v) { co_yield e; }
5   };
6
7   auto x = lamb(a);
8   auto y = lamb(b);
9
10  while(not x.finished() or not y.finished()) {
11    if(not x.finished()) {
12      co_yield x.value();
13      x.resume();
14    }
15
16    if(not y.finished()) {
17      co_yield y.value();
18      y.resume();
19    }
20  }
21 }
```


Interleaving two std::vector s

- The promise from the coroutine.

```
1 struct promise_type {
2   int _val{};
3
4   Generator           get_return_object() { return Generator{this}; }
5   std::suspend_never  initial_suspend() noexcept { return {}; }
6   std::suspend_always final_suspend() noexcept { return {}; }
7   std::suspend_always yield_value(int v)
8   {
9     _val = v;
10    return {};
11  }
12
13  void return_void() noexcept {}
14  void unhandled_exception() noexcept {}
15 };
```


© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Interleaving two std::vector s

- A generator for our coroutine function interleaved.

```
1 // struct Generator {
2 using Handle = std::coroutine_handle<promise_type>;
3 Handle mCoroHdl{};
4
5 explicit Generator(promise_type* p) noexcept : mCoroHdl{Handle::from_promise(*p)} {}
6
7 Generator(Generator&& rhs) noexcept : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)} {}
8
9 ~Generator() noexcept
10 {
11   if(mCoroHdl) { mCoroHdl.destroy(); }
12 }
13
14 int value() const { return mCoroHdl.promise()._val; }
15
16 bool finished() const { return mCoroHdl.done(); }
17
18 void resume()
19 {
20   if(not finished()) { mCoroHdl.resume(); }
21 }
```

Interleaving two std::vector s

- How to use interleaved.

```
1 void Use()
2 {
3   std::vector a{2, 4, 6, 8};
4   std::vector b{3, 5, 7, 9};
5
6   Generator g{interleaved(std::move(a), std::move(b))};
7
8   while(not g.finished()) {
9     std::cout << g.value() << '\n';
10
11     g.resume();
12   }
13 }
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Next task: Plastic surgeon required! I'm sure we all would like to use a range-based for-loop instead of `while`!

Interleaving two `std::vector` s - Beautification

- Adding support for range-based for loops et. al.
  - We need an iterator which fullfils the iterator-concept: equal comparable, incrementable, dereferenceable.
  - This type is declared inside `Generator`, but you're free to write a more general version.

```
1 struct sentinel {};
2
3 struct iterator {
4   Handle mCoroHdl{};
5
6   bool operator==(sentinel) const
7   {
8     return mCoroHdl.done();
9   }
10
11   iterator& operator++()
12   {
13     mCoroHdl.resume();
14     return *this;
15   }
16
17   const int operator*() const
18   {
19     return mCoroHdl.promise()._val;
20   }
21 };
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Interleaving two `std::vector` s - Beautification

- Adding support for the iterator to `Generator` of the coroutine.

```
1 // struct Generator {
2 // ...
3 iterator begin() { return {mCoroHdl}; }
4 sentinel end() { return {}; }
5 // };
```

```
1 std::vector a{2, 4, 6, 8};
2 std::vector b{3, 5, 7, 9};
3
4 Generator g{interleaved(std::move(a), std::move(b))};
5
6 for(const auto& e : g) { std::cout << e << '\n'; }
```


Another task: Scheduling multiple tasks.

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Cooperative vs. preemptive multitasking

With preemptive multitasking, the thread has no control over:

- when it runs,
- on which CPU or,
- for how long.

In cooperative multitasking, the thread decides:

- how long it runs, and
- when it is time to give control to another thread.

- Instead of using locks as in preemptive multitasking, we say **`co_yield`** or **`co_await`**.

Scheduling multiple tasks

- Starting and scheduling two tasks.

```
1 void Use()
2 {
3   Scheduler scheduler{};
4
5   taskA(scheduler);
6   taskB(scheduler);
7
8   while(scheduler.schedule()) {}
9 }
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Scheduling multiple tasks

- Two exemplary tasks.
- To suspend execution a task must call `co_await` reaching into the scheduler.

```
1 Task taskA(Scheduler& sched)
2 {
3   std::cout << "Hello, from task A\n";
4
5   co_await sched.suspend();
6
7   std::cout << "a is back doing work\n";
8
9   co_await sched.suspend();
10
11  std::cout << "a is back doing more work\n";
12 }
```

```
1 Task taskB(Scheduler& sched)
2 {
3   std::cout << "Hello, from task B\n";
4
5   co_await sched.suspend();
6
7   std::cout << "b is back doing work\n";
8
9   co_await sched.suspend();
10
11  std::cout << "b is back doing more work\n";
12 }
```


Scheduling multiple tasks

- The Scheduler.

```
1 struct Scheduler {
2   std::list<std::coroutine_handle<>> _tasks{};
3
4   bool schedule()
5   {
6     auto task = _tasks.front();
7     _tasks.pop_front();
8
9     if(not task.done()) { task.resume(); }
10
11    return not _tasks.empty();
12  }
13
14  auto suspend()
15  {
16    struct awaiter : std::suspend_always {
17      Scheduler& _sched;
18
19      explicit awaiter(Scheduler& sched) : _sched{sched} {}
20      void await_suspend(std::coroutine_handle<> coro) const noexcept { _sched._tasks.push_back(coro); }
21    };
22
23    return awaiter{*this};
24  }
25 };
```


© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Scheduling multiple tasks

- The `Task` type holding the coroutines `promise_type`.

```
1 struct Task {
2   struct promise_type {
3     Task              get_return_object() noexcept { return {}; }
4     std::suspend_never initial_suspend() noexcept { return {}; }
5     std::suspend_never final_suspend() noexcept { return {}; }
6     void              return_void() noexcept {}
7     void              unhandled_exception() noexcept {}
8   };
9 };
```

Scheduling multiple tasks - an alternative

- Starting and scheduling two tasks. This time using a global object.

```
1 void Use()
2 {
3   taskA();
4   taskB();
5
6   while(gScheduler.schedule()) {}
7 }
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Scheduling multiple tasks - an alternative

- Two exemplary tasks.
- To suspend execution a task must say `co_await` this time calling the `operator co_await` of an independent type `suspend`.

```
1  Task taskA()
2  {
3    std::cout << "Hello, from task A\n";
4
5    co_await suspend{};
6
7    std::cout << "a is back doing work\n";
8
9    co_await suspend{};
10
11   std::cout << "a is back doing more work\n";
12 }
```

```
1  Task taskB()
2  {
3    std::cout << "Hello, from task B\n";
4
5    co_await suspend{};
6
7    std::cout << "b is back doing work\n";
8
9    co_await suspend{};
10
11   std::cout << "b is back doing more work\n";
12 }
```

Scheduling multiple tasks - an alternative

- The Scheduler.

```
1  struct Scheduler {
2    std::list<std::coroutine_handle<>> _tasks{};
3
4    void suspend(std::coroutine_handle<> coro) { _tasks.push_back(coro); }
5
6    bool schedule()
7    {
8      auto task = _tasks.front();
9      _tasks.pop_front();
10
11     if(not task.done()) { task.resume(); }
12
13     return not _tasks.empty();
14   }
15 };
```

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

Scheduling multiple tasks - an alternative

- The Task type holding the coroutines promise_type.

```
1 static Scheduler gScheduler{};
2
3 struct suspend {
4   auto operator co_await()
5   {
6     struct awaiter : std::suspend_always {
7       void await_suspend(std::coroutine_handle<> coro) const noexcept { gScheduler.suspend(coro); }
8     };
9
10    return awaiter{};
11  }
12 };
```

Coroutine restrictions

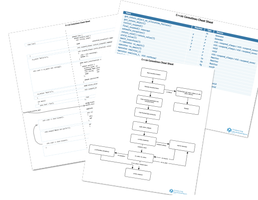
- There are some limitations in which functions can be a coroutine and what they must look like.
  - **constexpr**-functions cannot be coroutines. Subsequently, this is true for **consteval**-functions.
  - Neither a constructor nor a destructor can be a coroutine.
  - A function using varargs. A variadic function template works.
  - A function with plain **auto** as a return-type or with a concept type cannot be a coroutine. **auto** with trailing return-type works.
  - Further, a coroutine cannot use plain **return**. It must be either **co_return** or **co_yield**.
  - And last but not least, main cannot be a coroutine.
- Lambdas, on the other hand, can be coroutines.

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

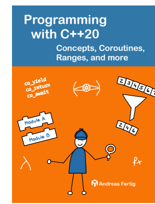
}

I am Fertig.

C++20 Coroutine Cheat Sheet



fertig.to/subscribe



fertig.to/bpwcpp20

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++ 11.1.0
  - clang version 14.0.0 (https://github.com/tru/llvm-release-build fc075d7c96fe7c992dde351695a5d25fe084794a)

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com

References

[1] Knuth D., *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Pearson Education, 1997.

Images:
37: Franziska Panter

Upcoming Events

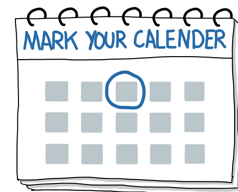
Talks

- *C++ Coroutines from scratch*, NDC TechTown, September 20

Training Classes

- *Programming with C++20*, CppCon, September 27 - 30
- *Modern C++: When Efficiency Matters*, CppCon, October 09 - 12

For my upcoming talks you can check https://andreasfertig.com/talks/.
For my courses you can check https://andreasfertig.com/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.com/newsletter/.



About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 23.
Andreas is involved in the C++ standardization committee, in which the new standards are developed. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus understand constructs even better.
Before working as a trainer and consultant, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
You can find Andreas online at andreasfertig.com.

Andreas Fertig v1.0 C++ Coroutines from scratch 38

© 2023 Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com