C++20's Coroutines for Beginners

Presentation Material

C++Online, Online, 2024-03-01

Andreas Fertig

Write unique code!

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

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.

C++20's Coroutines for Beginners

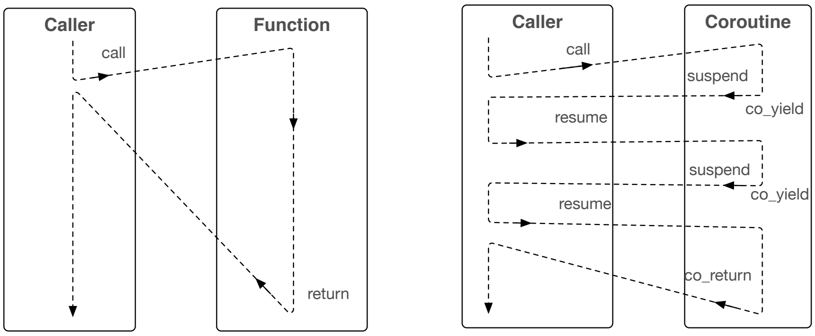
Andreas Fertig
https://AndreasFertig.com
post@AndreasFertig.com
fertig.to/@social

fertig

adjective /ˈfɛrtɪç/

finished
ready
complete
completed

Function vs. Coroutine comparison



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.

https://AndreasFertig.com
post@AndreasFertig.com

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

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.

Coroutine chat

```cpp
Chat Fun()  // A Wrapper type Chat containing the promise type
{
  co_yield "Hello!\n"s;  // B Calls promise_type.yield_value

  std::cout << co_await std::string{};  // C Calls promise_type.await_transform

  co_return "Here!\n"s;  // D Calls promise_type.return_value
}

void Use()
{
  Chat marco = Fun();  // E Creation of the coroutine

  std::cout << marco.listen();  // F Trigger the machine

  marco.answer("Where are you?\n"s);  // G Send data into the coroutine

  std::cout << marco.listen();  // H Wait for more data from the coroutine
}
```

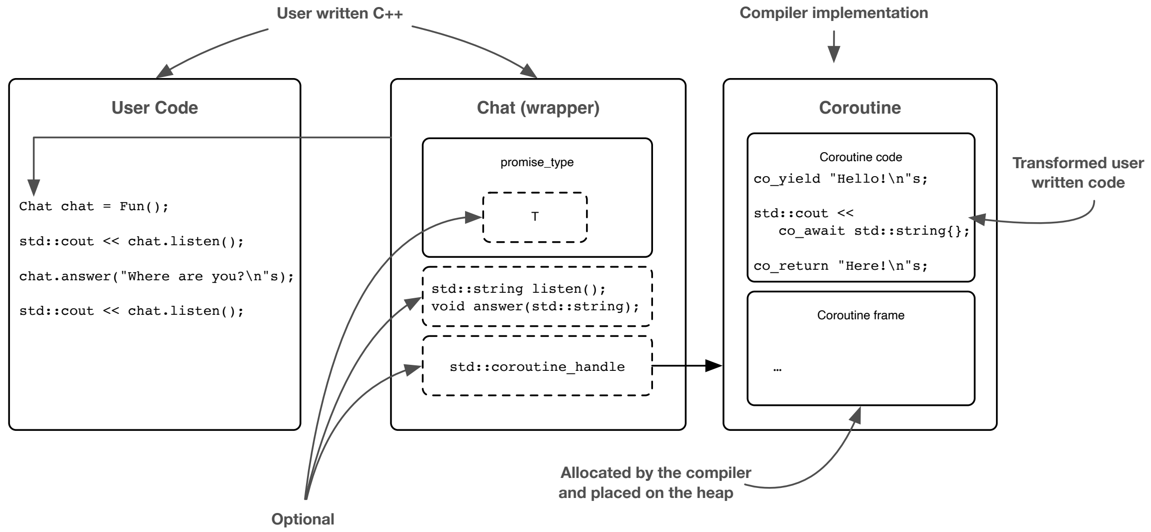
Coroutine chat

```
struct promise_type {
  std::string _msgOut{}, _msgIn{};   (A) Storing a value from or for the coroutine

  void                unhandled_exception() noexcept {}                    (B) What to do in case of an exception
  Chat                get_return_object() { return Chat{*this}; }          (C) Coroutine creation
  std::suspend_always initial_suspend() noexcept { return {}; }            (D) Startup
  std::suspend_always yield_value(std::string msg) noexcept               (F) Value from co_yield
  {
    _msgOut = std::move(msg);
    return {};
  }

  auto await_transform(std::string) noexcept   (G) Value from co_await
  {
    struct awaiter {   (H) Customized version instead of using suspend_always or suspend_never
      promise_type&  pt;
      constexpr bool await_ready() const noexcept { return true; }
      std::string    await_resume() const noexcept { return std::move(pt._msgIn); }
      void           await_suspend(std::coroutine_handle<>) const noexcept {}
    };

    return awaiter{*this};
  }

  void return_value(std::string msg) noexcept { _msgOut = std::move(msg); }   (I) Value from co_return
  std::suspend_always final_suspend() noexcept { return {}; }                 (E) Ending
};
```


Coroutine chat

```
struct Chat {
#include "promise-type.h"   // Don't do that at work!

  std::coroutine_handle<promise_type> mHandle{};   (A)

  explicit Chat(promise_type& p)
  : mHandle{std::coroutine_handle<promise_type>::from_promise(p)} {}   (B) Get the handle form the promise

  Chat(Chat&& rhs) noexcept : mHandle{std::exchange(rhs.mHandle, nullptr)} {}   (C) Move only!

  ~Chat() noexcept   (D) Care taking, destroying the handle if needed
  {
    if(mHandle) { mHandle.destroy(); }
  }

  std::string listen()   (E) Activate the coroutine and wait for data.
  {
    if(not mHandle.done()) { mHandle.resume(); }
    return std::move(mHandle.promise()._msgOut);
  }

  void answer(std::string msg)   (F) Send data to the coroutine and activate it.
  {
    mHandle.promise()._msgIn = std::move(msg);
    if(not mHandle.done()) { mHandle.resume(); }
  }
};
```

Coroutine customization points



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`).

https://AndreasFertig.com
post@AndreasFertig.com

Helper types for Coroutines

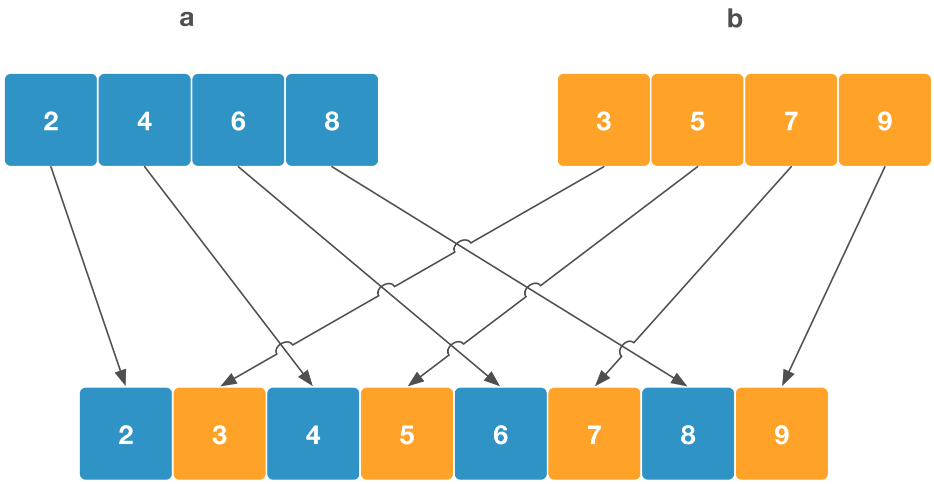
- For `yield_value`, `initial_suspend`, `final_suspend`, as well as **`co_await`** / `await_transform`, we have two helper types in the Standard Template Library (STL):
  - `std::suspend_always`: The method `await_ready` always returns **`false`**, indicating that an await expression always suspends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

```cpp
struct suspend_always {
  constexpr bool await_ready() const noexcept
  {
    return  false ;
  }

  constexpr void
  await_suspend(std::coroutine_handle<>) const noexcept
  {}

  constexpr void await_resume() const noexcept {}
};
```

```cpp
struct suspend_never {
  constexpr bool await_ready() const noexcept
  {
    return  true ;
  }

  constexpr void
  await_suspend(std::coroutine_handle<>) const noexcept
  {}

  constexpr void await_resume() const noexcept {}
};
```


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

Interleaving two `std::vector` s



Interleaving two `std::vector` s

- The interleave coroutine function.

```
Generator interleaved(std::vector<int> a, std::vector<int> b)
{
  auto lamb = [](std::vector<int>& v) -> Generator {
    for(const auto& e : v) { co_yield e; }
  };

  auto x = lamb(a);
  auto y = lamb(b);

  while(not x.finished() or not y.finished()) {
    if(not x.finished()) {
      co_yield x.value();
      x.resume();
    }

    if(not y.finished()) {
      co_yield y.value();
      y.resume();
    }
  }
}
```

Interleaving two `std::vector` s

- The promise from the coroutine.

```
struct promise_type {
  int _val{};

  Generator           get_return_object() { return Generator{*this}; }
  std::suspend_never  initial_suspend() noexcept { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }
  std::suspend_always yield_value(int v)
  {
    _val = v;
    return {};
  }

  void return_void() noexcept {}
  void unhandled_exception() noexcept {}
};
```


Interleaving two `std::vector` s

- A generator for our coroutine function `interleaved`.

```
// struct Generator {
std::coroutine_handle<promise_type> mHandle{};

explicit Generator(promise_type& p) noexcept : mHandle{std::coroutine_handle<promise_type>::from_promise(p)} {}

Generator(Generator&& rhs) noexcept : mHandle{std::exchange(rhs.mHandle, nullptr)} {}

~Generator() noexcept
{
  if(mHandle) { mHandle.destroy(); }
}

int value() const { return mHandle.promise()._val; }

bool finished() const { return mHandle.done(); }

void resume()
{
  if(not finished()) { mHandle.resume(); }
}
```

Interleaving two `std::vector` s

- How to use `interleaved`.

```cpp
void Use()
{
  std::vector a{2, 4, 6, 8};
  std::vector b{3, 5, 7, 9};

  Generator g{interleaved(std::move(a), std::move(b))};

  while(not g.finished()) {
    std::cout << g.value() << '\n';

    g.resume();
  }
}
```


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

https://AndreasFertig.com
post@AndreasFertig.com

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.

```
struct iterator {
  std::coroutine_handle<promise_type> mHandle{};

  bool operator==(std::default_sentinel_t) const
  {
    return mHandle.done();
  }

  iterator& operator++()
  {
    mHandle.resume();
    return *this;
  }

  const int operator*() const
  {
    return mHandle.promise()._val;
  }
};
```

Andreas Fertig v1.0 C++20's Coroutines for Beginners 22

Interleaving two `std::vector` s - Beautification

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

```
// struct Generator {
// ...
iterator                begin() { return {mHandle}; }
std::default_sentinel_t end() { return {}; }
// };
```

```
std::vector a{2, 4, 6, 8};
std::vector b{3, 5, 7, 9};

Generator g{interleaved(std::move(a), std::move(b))};

for(const auto& e : g) { std::cout << e << '\n'; }
```

Andreas Fertig v1.0 C++20's Coroutines for Beginners 23

https://AndreasFertig.com
post@AndreasFertig.com

Another task:
Scheduling multiple tasks.

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.

```
void Use()
{
  Scheduler scheduler{};

  taskA(scheduler);
  taskB(scheduler);

  while(scheduler.schedule()) {}
}
```


Scheduling multiple tasks

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

```
Task taskA(Scheduler& sched)
{
  std::cout << "Hello, from task A\n"sv;

  co_await sched.suspend();

  std::cout << "a is back doing work\n"sv;

  co_await sched.suspend();

  std::cout << "a is back doing more work\n"sv;
}
```

```
Task taskB(Scheduler& sched)
{
  std::cout << "Hello, from task B\n"sv;

  co_await sched.suspend();

  std::cout << "b is back doing work\n"sv;

  co_await sched.suspend();

  std::cout << "b is back doing more work\n"sv;
}
```

Scheduling multiple tasks

- The Scheduler.

```
struct Scheduler {
  std::list<std::coroutine_handle<>> _tasks{};

  bool schedule()
  {
    auto task = _tasks.front();
    _tasks.pop_front();

    if(not task.done()) { task.resume(); }

    return not _tasks.empty();
  }

  auto suspend()
  {
    struct awaiter : std::suspend_always {
      Scheduler& _sched;

      explicit awaiter(Scheduler& sched) : _sched{sched} {}
      void await_suspend(std::coroutine_handle<> coro) const noexcept { _sched._tasks.push_back(coro); }
    };

    return awaiter{*this};
  }
};
```


Scheduling multiple tasks

- The Task type holding the coroutines promise_type.

```
struct Task {
  struct promise_type {
    Task              get_return_object() noexcept { return {}; }
    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void              return_void() noexcept {}
    void              unhandled_exception() noexcept {}
  };
};
```

Scheduling multiple tasks - an alternative

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

```cpp
void Use()
{
  taskA();
  taskB();

  while(gScheduler.schedule()) {}
}
```


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.

```cpp
Task taskA()
{
  std::cout << "Hello, from task A\n"sv;

  co_await suspend{};

  std::cout << "a is back doing work\n"sv;

  co_await suspend{};

  std::cout << "a is back doing more work\n"sv;
}
```

```cpp
Task taskB()
{
  std::cout << "Hello, from task B\n"sv;

  co_await suspend{};

  std::cout << "b is back doing work\n"sv;

  co_await suspend{};

  std::cout << "b is back doing more work\n"sv;
}
```

Scheduling multiple tasks - an alternative

- The Scheduler.

```
struct Scheduler {
  std::list<std::coroutine_handle<>> _tasks{};

  void suspend(std::coroutine_handle<> coro) { _tasks.push_back(coro); }

  bool schedule()
  {
    auto task = _tasks.front();
    _tasks.pop_front();

    if(not task.done()) { task.resume(); }

    return not _tasks.empty();
  }
};
```


Scheduling multiple tasks - an alternative

- The **operator co_await** interacting with the scheduler.

```
static Scheduler gScheduler{};

struct suspend {
  auto operator co_await()
  {
    struct awaiter : std::suspend_always {
      void await_suspend(std::coroutine_handle<> coro) const noexcept { gScheduler.suspend(coro); }
    };

    return awaiter{};
  }
};
```

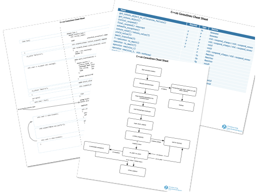
Coroutine restrictions

- 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.

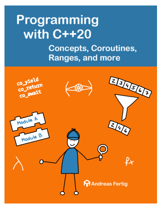
}

I am Fertig.

C++20 Coroutine Cheat Sheet



fertig.to/subscribe



fertig.to/bpwcpp20

https://AndreasFertig.com
post@AndreasFertig.com

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - GCC 13.2.0
  - Clang 17.0.0

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/

References

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

Images:
38: Franziska Panter

Upcoming Events

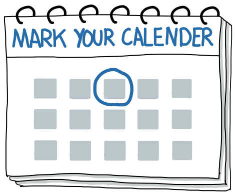
Talks

- *C++20 Coroutinen - Ein Einstieg*, ADC, May 07

Training Classes

- *A day with coroutines*, C++Online, March 04
- *C++20 Coroutinen - Ein Einstieg*, ADC, May 06

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 consultant for C++ for standards 11 to 23.
Andreas is involved in the C++ standardization committee, developing the new standards. 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 training and consulting, 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.

https://AndreasFertig.com
post@AndreasFertig.com