Einführung in C++20 Coroutinen

Präsentationsunterlagen

ESE Kongress, Sindelfingen, 12.09.2022

Andreas Fertig

Write unique code!

Version: v1.0

Planung, Satz und Einbandentwurf: Andreas Fertig
Titelbild und Illustrationen: Franziska Panter, franziskapanter.com
Herstellung und Verlag: Andreas Fertig

Stil und Konventionen

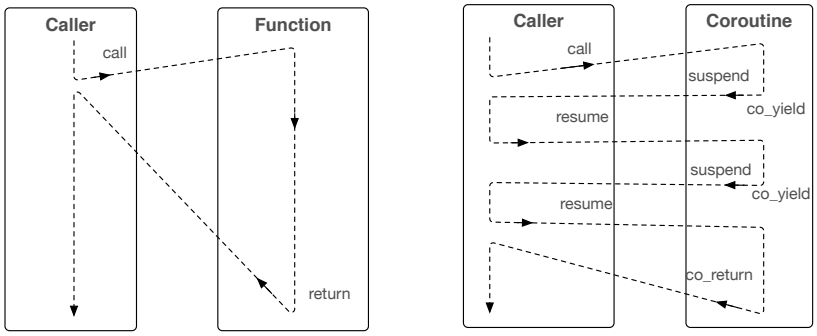
Im Folgenden wird die Ausführung eines Programms gezeigt. Ich habe hier den Linux-Weg verwendet und die Angabe des gewünschten Ausgabenamens übersprungen, was zu `a.out` als Programmname führt.

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

- `<string>` steht für eine Header-Datei mit dem Namen `string`
- `[[xyz]]` markiert ein C++-Attribut mit dem Namen `xyz`.

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

Vergleich von Funktionen und Coroutinen



Was sind Coroutinen?

- Der Begriff Coroutine ist in der Informatik seit seiner ersten Prägung im Jahr 1958 durch Melvin Conway fest verankert [1].
- Es existieren zwei verschiedene Formen:
  - Stack*full*
  - Stack*less* (die Form welche wir in C++ haben)
- Stackless bedeutet, dass die Daten einer Coroutine, das Coroutine-Frames, auf dem Heap gespeichert werden.
- Wir sprechen von kooperativem Multitasking bei der Verwendung von Coroutinen.
- Coroutinen können Ihren Code vereinfachen!
  - Wir können einige Funktionszeiger (Callbacks) durch Coroutinen ersetzen.
  - Parser sind mit Coroutinen viel besser lesbar.
  - Es ist kein Code für die Zustandsverwaltung erforderlich, da die Coroutine die Buchhaltung übernimmt.

Elemente einer Coroutine

- In C++ besteht eine Coroutine aus:
  - Ein Wrapper-Typ. Dies ist der Rückgabetyp des Funktionsprototyps der Coroutine.
    - Mit diesem Datentyp kann die Coroutine von außen gesteuert werden. Zum Beispiel die Wiederaufnahme der Coroutine oder Abrufen von Daten in oder von der Coroutine durch Speichern eines Handles für die Coroutine im Wrapper-Typ.
  - Der Compiler sucht im Rückgabedatentyp der Coroutine (dem Wrapper-Typ) nach einem *Datentyp* mit dem genauen Namen `promise_type`. Das ist die Kontrolle von innen.
    - Dieser Datentyp kann ein Typalias sein, oder
    - ein **`typedef`**,
    - oder Sie können den Datentyp direkt innerhalb des Coroutine-Wrapper-Typs deklarieren.
  - Ein *awaitable* Typ, der ins Spiel kommt, sobald wir **`co_await`** verwenden.
  - Wir verwenden auch oft einen anderen Teil, einen Iterator.
- Eine Coroutine in C++ ist ein Zustandsautomat, der durch den `promise_type` gesteuert und angepasst werden kann.
- Die eigentliche Coroutinen-Funktion verwendet, **`co_yield`**, **`co_await`** oder **`co_return`** für die Kommunikation mit der Außenwelt verwendet.

Disclaimer

Bitte beachten Sie, dass ich versucht habe, den Code, den Sie sehen, so einfach wie möglich zu halten. Fokus auf Coroutinen. Im Produktionscode arbeite ich mehr mit **`public`** und **`private`** sowie potenziellen Gettern und Settern. Darüber hinaus verwende ich im Produktionscode viel generischen Code, um Wiederholungen gering zu halten.
Mein Ziel ist es, Ihnen zu helfen, Coroutinen zu verstehen. Ich bin zuversichtlich, dass Sie den Code, den Sie sehen werden, mit den üblichen C++-Best Practices verbessern können.
Ich deklariere außerdem nie mehr als eine Variable pro Zeile … Foliencode ist die einzige Ausnahme.

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 };
```

Coroutine chat

```
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 };
```

Beispiel:
Schedulen mehrerer Tasks

Schedulen mehrerer Tasks

- Zwei Aufgaben starten und schedulen.

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

  taskA(scheduler);
  taskB(scheduler);

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


Schedulen mehrerer Tasks

- Zwei beispielhafte Aufgaben.
- Um die Ausführung einer Aufgabe anzuhalten, muss sie **`co_await`** aufrufen und in den Scheduler gelangen.

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

  co_await sched.suspend();

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

  co_await sched.suspend();

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

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

  co_await sched.suspend();

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

  co_await sched.suspend();

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

Schedulen mehrerer Tasks

- Der Scheduler

```cpp
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};
  }
};
```


Schedulen mehrerer Tasks

- Der `Task`-Datentyp, der den Coroutinen `promise_type` enthält.

```cpp
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 {}
  };
};
```

Beispiel: Schedulen mehrerer Tasks - Alternative Implementierung

Schedulen mehrerer Tasks - Eine Alternative

- Zwei Aufgaben starten und schedulen. Diesmal mit einem globalen Objekt.

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

Schedulen mehrerer Tasks - Eine Alternative

- Zwei beispielhafte Aufgaben.
- Um die Ausführung einer Aufgabe anzuhalten, muss sie **co_await** sagen, dieses Mal wird der **operator co_await** eines unabhängigen Typs suspend aufgerufen.

```
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 }
```


Schedulen mehrerer Tasks - Eine Alternative

- Der 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 };
```

Schedulen mehrerer Tasks - Eine Alternative

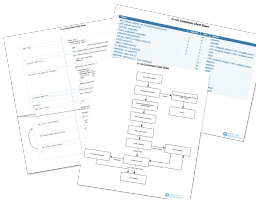
- Der `Task`-Datentyp, der den Coroutinen `promise_type` enthält.

```
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 };
```

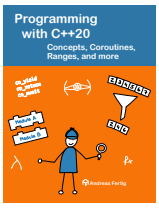

}

Ich bin Fertig.

C++20 Coroutine Cheat Sheet



fertig.to/subscribe



fertig.to/bpwcpp20

Verwendete Compiler & Typografie

Verwendete Compiler

- Compiler welche zum Übersetzen des (meisten) Codes verwendet wurden.
  - 13.2.0
  - 17.0.0

Typografie

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

Quellen

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

Bilder:

22: Franziska Panter

Nächste Events

Zukünftige Vorträge: https://andreasfertig.com/de/vortraege/.
Mehr zu meinen Schulungsangeboten gibt es hier: https://andreasfertig.com/de/schulungen/.
Immer aktuell Informiert? Hier geht es zum Newsletter: https://andreasfertig.com/newsletter/.

Über Andreas Fertig

Foto: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, Geschäftsführer der Unique Code GmbH, ist erfahrener Trainer und Dozent für C++ für die Standards 11 bis 23.
Andreas engagiert sich im C++ Standardisierungskomitee, in dem die neuen Standards erarbeitet werden. Auf internationalen Konferenzen präsentiert er, wie sich Code besser schreiben lässt. Er publiziert Fachartikel z.B. für das iX Magazin und hat mehrere Fachbücher zu C++ veröffentlicht.
Mit dem Tool C++ Insights (https://cppinsights.io) hat Andreas ein international anerkanntes Werkzeug geschaffen, das Programmierer:innen ermöglicht, C++ noch besser zu verstehen.
Vor seiner Tätigkeit als Trainer und Berater arbeitete er zehn Jahre für die Philips Medizin Systeme GmbH als C++ Softwareentwickler und Architekt mit Schwerpunkt auf eingebetteten Systemen.
Andreas ist online unter andreasfertig.com erreichbar.