Coroutines

The future of `future`

Core C++ Conf. 2019

Yehezkel Bernat

YehezkelShB@gmail.com, @YehezkelShB

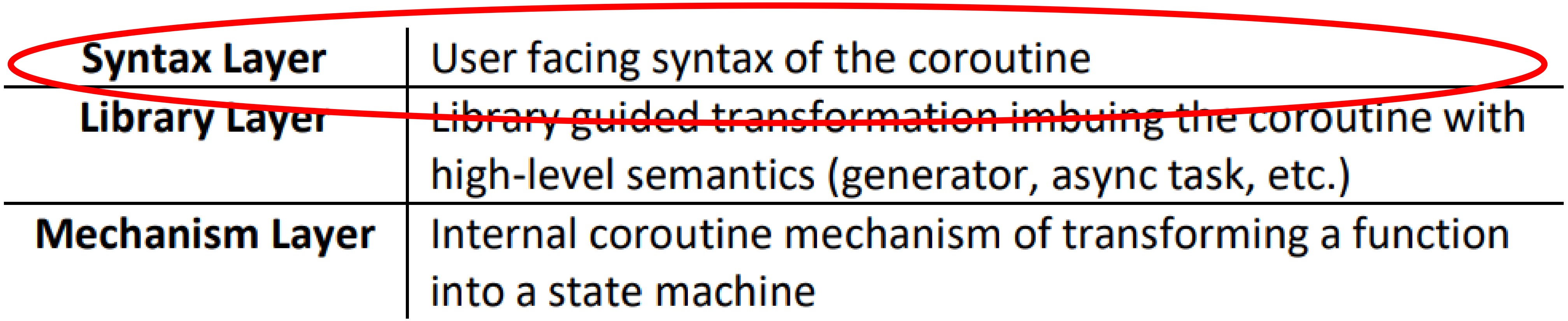
Playground Rules / Assumptions

- I assume we all know about (at least the basics of) multithreading
- We all feel comfortable with using:
  - `auto` for type deduction
  - Lambda expressions
- Please ask questions

Intro

- Today we'll learn about coroutines
- Published as a TS in 2017-12-05
- Merged for C++20 in Kona (Feb ‘19)
- Implemented in MSVC
- Implemented in Clang
- gcc implementation started
- More than 4 years of usage in production code (on Windows; 3y on Linux)

Agenda



Syntax Layer	User facing syntax of the coroutine
Library Layer	Library guided transformation imbuing the coroutine with high-level semantics (generator, async task, etc.)
Mechanism Layer	Internal coroutine mechanism of transforming a function into a state machine

- What we are NOT going to discuss:
  - How the compiler transforms our code
  - How to write a library type for using coroutines
  - (See the links in the references for this kind of info)
- We'll focus on the user side

https://wg21.link/p1362r0 by Gor Nishanov

co_await

Let’s do some I/O

```
   std::vector<std::string>
              readLines(std::string path);
   uint64_t
countLines(std::vector<std::string> paths);
```

Let’s do some I/O

```
std::future<std::vector<std::string>>
                       readLines(std::string path);
std::future<uint64_t>
         countLines(std::vector<std::string> paths);
```

Let’s do some I/O

```
std::future<std::vector<std::string>> readLines(std::string path);
std::future<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
    auto f1 = readLines(argv[1]);
    auto f2 = countLines(getFileList());
    // ...
    for (const auto& line : f1.get())
                std::cout << line << '\n';
    std::cout << f2.get() << '\n';
}
```

Composite the functions

```
std::future<std::vector<std::string>> readLines(std::string path);
std::future<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
    auto f1 = readLines(argv[1]);
    auto f2 = countLines(f1.get());
    // ...
    std::cout << f2.get() << '\n’;
}
```

- Works, but first function runs synchronously

Let’s try .get() again

```
int main(int argc, char* argv[]) {
    auto f =                                [argv] {
        auto f1 = readLines(argv[1]);
        auto f2 = countLines(f1.get());
        return f2.get();
    } ;
    // ...
    std::cout << f.get() << '\n’;
}
```

Let’s try `.get()` again

```
int main(int argc, char* argv[]) {
    auto f = std::async(std::launch::async, [argv] {
        auto f1 = readLines(argv[1]);
        auto f2 = countLines(f1.get());
        return f2.get();
    });
    // ...
    std::cout << f.get() << '\n’;
}
```

- Works, and even fully async, but what a waste of a thread

Enter cppcoro::task<T>

```
cppcoro::task<std::vector<std::string>>
                        readLines(std::string path);
cppcoro::task<uint64_t>
         countLines(std::vector<std::string> paths);
```

```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]()                             {
                             paths =           readLines(argv[1]);
                             lines =           countLines(paths);
       return lines;
  };



}
```

```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
                             paths =          readLines(argv[1]);
                             lines =          countLines(paths);
       return lines;
  };





}
```

```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    std::vector<std::string> paths = co_await readLines(argv[1]);
                             lines =          countLines(paths);
       return lines;
  };



}
```

```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    std::vector<std::string> paths = co_await readLines(argv[1]);
    uint64_t                 lines = co_await countLines(paths);
       return lines;
  };





}
```

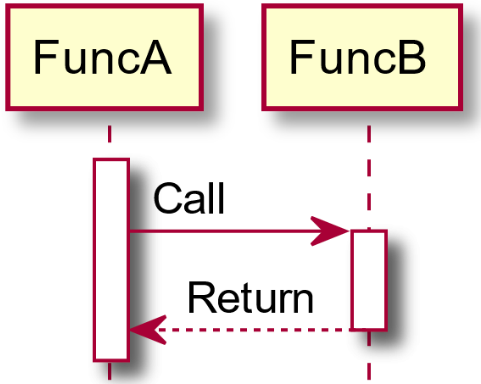
```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    std::vector<std::string> paths = co_await readLines(argv[1]);
    uint64_t                 lines = co_await countLines(paths);
    co_return lines;
  };



}
```

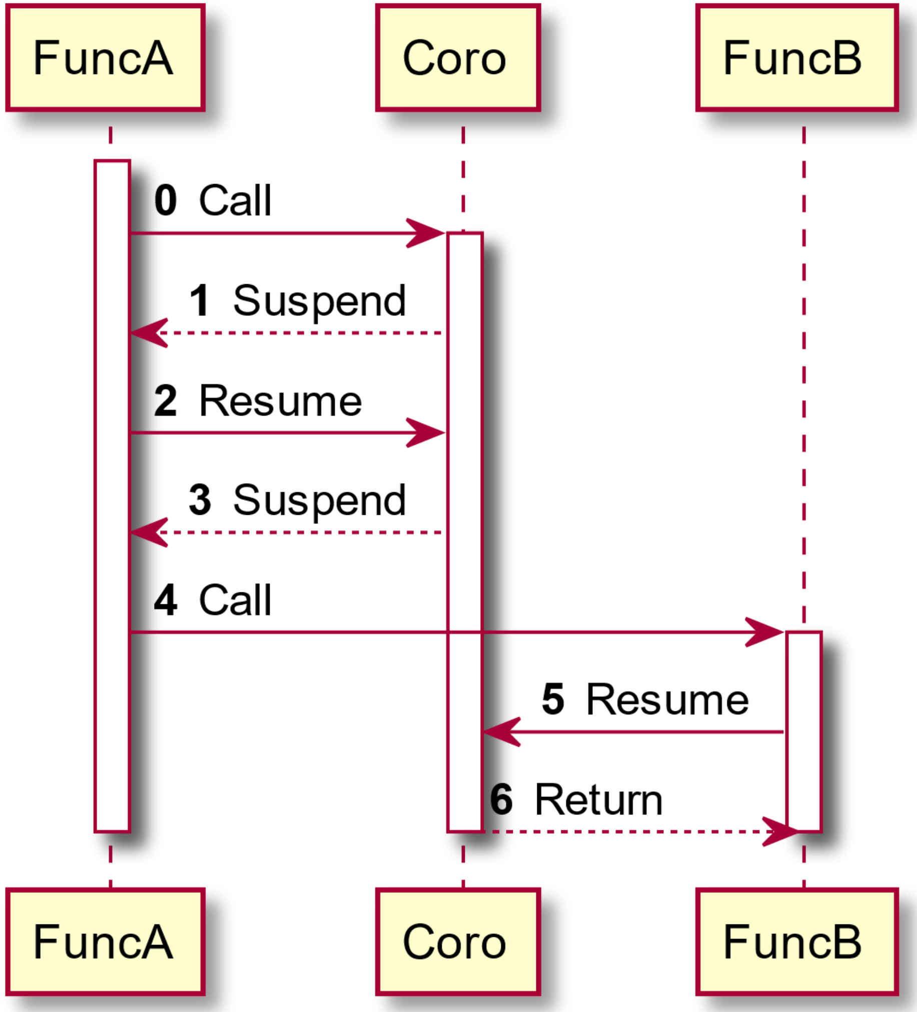
```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    std::vector<std::string> paths = co_await readLines(argv[1]);
    uint64_t                 lines = co_await countLines(paths);
    co_return lines;
  };
  auto t2 = []() -> cppcoro::task<void> { // Rest of main() here
    co_return; // or have co_await somewhere
  };
                                     cppcoro::when_all(t1(), t2()) ;

}
```

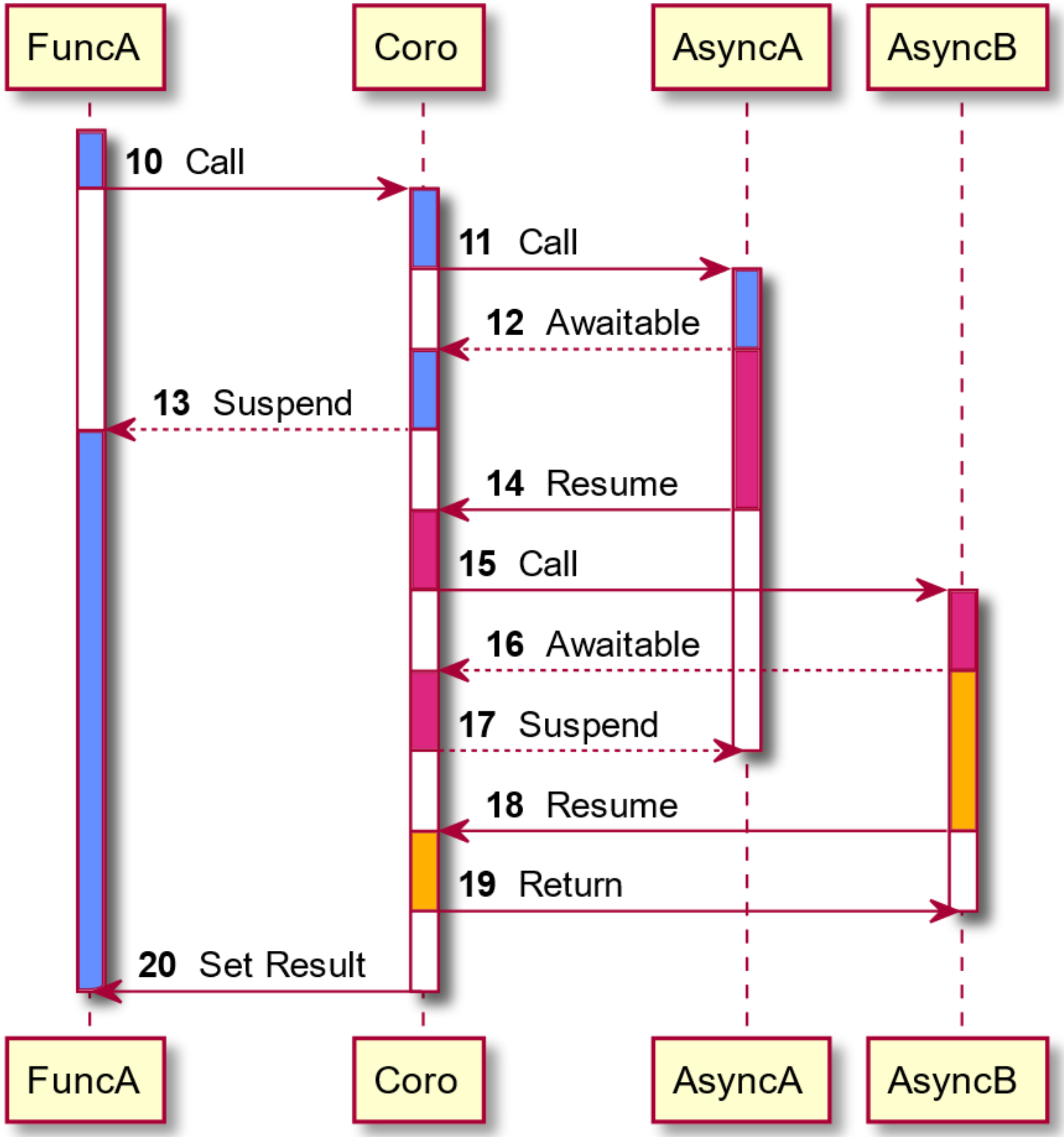
```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    std::vector<std::string> paths = co_await readLines(argv[1]);
    uint64_t                 lines = co_await countLines(paths);
    co_return lines;
  };
  auto t2 = []() -> cppcoro::task<void> { // Rest of main() here
    co_return; // or have co_await somewhere
  };
                 cppcoro::sync_wait(cppcoro::when_all(t1(), t2()));

}
```

```
cppcoro::task<std::vector<std::string>> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::vector<std::string> paths);
int main(int argc, char* argv[]) {
  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    std::vector<std::string> paths = co_await readLines(argv[1]);
    uint64_t                 lines = co_await countLines(paths);
    co_return lines;
  };
  auto t2 = []() -> cppcoro::task<void> { // Rest of main() here
    co_return; // or have co_await somewhere
  };
  auto results = cppcoro::sync_wait(cppcoro::when_all(t1(), t2()));
  std::cout << std::get<0>(results) << '\n';
}
```

Coroutine vs. Regular Function






FuncA
Coro
AsyncA
AsyncB
10 Call
11 Call
12 Awaitable
13 Suspend
14 Resume
15 Call
16 Awaitable
17 Suspend
18 Resume
19 Return
20 Set Result
FuncA
Coro
AsyncA
AsyncB

So what is a coroutine?

- A generalized form of a routine (normal function)
- With regular functions, we have two operations:
  - Call
  - Return
- With coroutines, there are two additional ones:
  - Suspend
  - Resume
- The compiler makes sure the state (arguments and local vars) is saved
- The library decides how to use it and gives semantics to the mechanism

Coroutines – what we have seen so far

- The return type matters – semantics, behavior, what `co_*` word is allowed
  - Which is why I call such types “coroutine types”
- Having one of the `co_*` words is what turns a function into a coroutine
- We use `co_await` on an *Awaitable* type to suspend the coroutine
  - If it isn’t ready yet
- Usually, the coroutine is resumed when the *Awaitable* is ready
  - On the thread that has set the *Awaitable* ready
- `co_return` is used to return from a coroutine
- Generally, every “coroutine type” is expected to be *Awaitable*, but not vice versa

The big picture: where are coroutines used?

- On the top of the call stack there is a “real async” function
- On the bottom – something that waits for the result
    - At least eventually (as always was)
    - Immediately in cppcoro case (or any lazy task implementation)
        - But it can start another task as soon as the first one suspends
- It’s the in-between that was made simpler with coroutines

co_yield

Can we improve `readLines()`?

- Isn’t it a waste of time and space to read the whole file at once?
- Each line can be used separately
- With a huge file, the waste is huge too, and it may just fail
- Can we get the lines one-by-one?
- Usually, we would expect this to involve creation of a new type, which will be iterable or at least allow getting the next line
- Can it be done in a single function without all the boilerplate?

cppcoro::generator<T>

```
cppcoro::generator<std::string>
                       readLines(std::string path) {
  std::ifstream file(path);
  std::string line;
  while (std::getline(file, line))
      co_yield line;
}
```

The consumer side

```
cppcoro::generator<std::string> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::string path);
```

```
auto t1 = [argv]() -> cppcoro::task<uint64_t> {
  uint64_t lines = 0;
                          readLines(argv[1])
               lines  = co_await countLines(path);
  co_return lines;
};
```

The consumer side

```
cppcoro::generator<std::string> readLines(std::string path);
cppcoro::task<uint64_t> countLines(std::string path);
```

```
auto t1 = [argv]() -> cppcoro::task<uint64_t> {
  uint64_t lines = 0;
  for (const auto& path : readLines(argv[1]))
               lines += co_await countLines(path);
  co_return lines;
};
```

`co_yield` – What we have seen

- Allows writing a full object as a simple function
- Suspends the function (and makes it a coroutine) just like `co_await`
- `co_await` – for bringing a value into the coroutine
- `co_yield` – for passing a value out of the coroutine
- Usually, used for a generator and thus forms a range
  - It interacts very well with `std::ranges`, but that is another story and shall be told another time
- The return type is still playing a major role with allowing `co_yield`, defining the semantics, and of course defining the interface
- Typically, the resumer is:
  - after `co_await` – the Awaitable (continuation)
  - after `co_yield` – the caller

Haven’t we lost something?

- `readLines()` isn’t async anymore

```
cppcoro::async_generator<std::string>
                              readLines(std::string path) {
  auto file = cppcoro::read_only_file::open(IoService::get(), path);
  auto buffer = std::string(4096, '\0’); std::string line;
  for (uint64_t offset = 0, fileSize = file.size(); offset < fileSize;) {
    const auto bytesToRead = static_cast<size_t>(std::min<uint64_t>(buffer.size(), fileSize - offset));
    const auto bytesRead = co_await file.read(offset, buffer.data(), bytesToRead);
    for (size_t i = 0; i < bytesRead;) {
      auto currentLineEndLocation = buffer.find('\n', i);
      if (currentLineEndLocation == buffer.npos) { line.append(buffer, i); break; }
      line.append(buffer, i, currentLineEndLocation - i);
      co_yield line; line.clear(); i = currentLineEndLocation + 1;
    }
    offset += bytesRead;
  }
}
```

The consumer side

```
cppcoro::task<uint64_t> countLines(std::string path);

  auto t1 = [argv]() -> cppcoro::task<uint64_t> {
    uint64_t lines = 0;
    for co_await (const auto& path :
                        readLines(argv[1]))
                 lines += co_await countLines(path);
    co_return lines;
  };
```

Summary

- We have learned the basics of coroutines
    - `co_await`, `co_yield` and `co_return`
    - Suspend and resume
    - The return type matters
    - No change to the function declaration!
- We have seen a taste of what cppcoro lib provides
- We have understood where coroutines (usually) fit into the big picture

Thank You!

https://xkcd.com/208/ - Regular Expressions

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Wait, forgot to escape a space. Wheeeeee[taptaptap]eeeeee.

Thank You!

```
while (!timeIsUp())
{
    co_await question();
    co_yield std::optional<Answer>{};
}
```

References

- Code examples: https://github.com/YehezkelShB/CoreCpp2019-Coroutines
- Gor Nishanov’s series of presentations in CppCon, starting with CppCon 2014 (“await 2.0, Stackless Resumable Functions”)
- Lewis Baker’s blog post series: https://lewissbaker.github.io/
- cppcoro lib: https://github.com/lewissbaker/cppcoro
- Meeting C++ 2018, “Coroutine TS: A new way of thinking”, Andreas Reischuck
  - Code: https://github.com/arBmind/2018-cogen-en
    - Especially code/co_statemachine
  - Video: https://www.youtube.com/watch?v=RL5oYUl5548

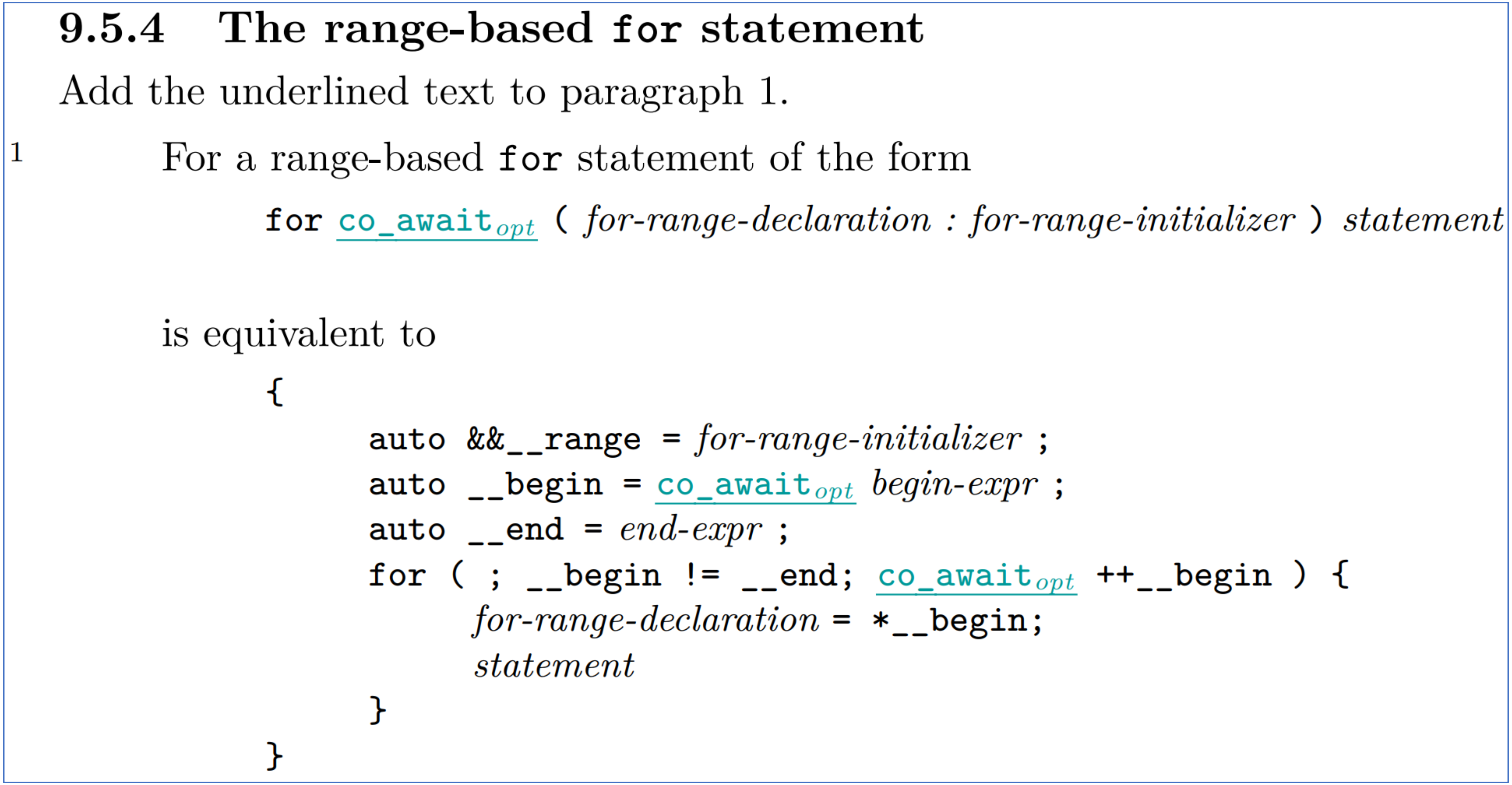
Backup

State machine in a function

```cpp
auto elevator = create();
for (auto event : getEventList())
{
    std::cout << "+ " << event << '\n';
    elevator.handle(event);
    std::cout << "@ " << elevator.state() << '\n';
}
```

Based on: Meeting C++ 2018, “Coroutine TS: A new way of thinking”, Andreas Reischuck

```
StateMachine<Event, State> create() {
    int curFloor = 0; int targetFloor = 0; State state = Idle{};
    auto matchers = overloaded{
        [](auto, AlarmPressed) -> State { return Broken{}; },
        [&](Idle, Called c) -> State {
            targetFloor = c.toFloor;
            return c.toFloor < curFloor ? Move::Down : Move::Up;
        } // ...
    };
    while (true) {
        Event event = co_yield state;
        state = std::visit(matchers, state, event);
    }
}
```



9.5.4 The range-based **`for`** statement

Add the underlined text to paragraph 1.

1 For a range-based **`for`** statement of the form

```
for co_await_opt ( for-range-declaration : for-range-initializer ) statement
```

is equivalent to

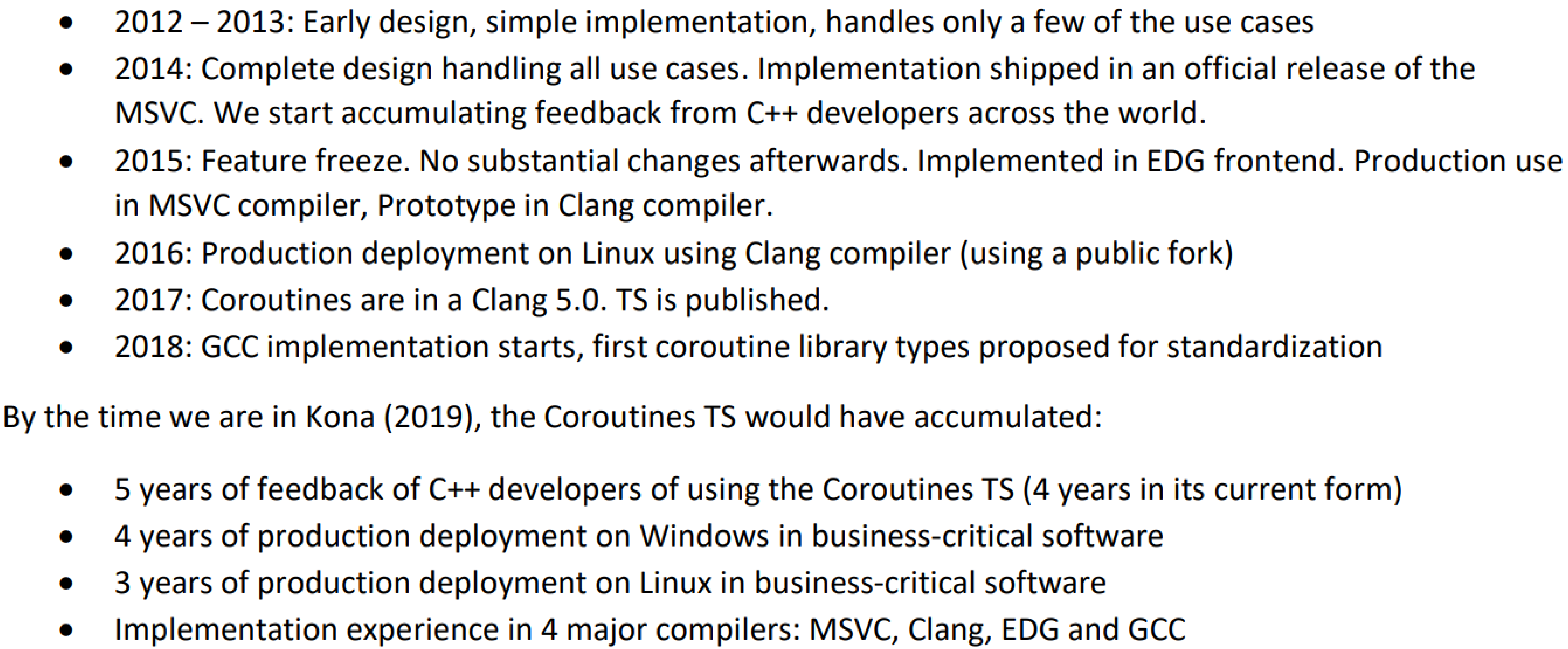
```
{
    auto &&__range = for-range-initializer ;
    auto __begin = co_await_opt begin-expr ;
    auto __end = end-expr ;
    for ( ; __begin != __end; co_await_opt ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Coroutines TS
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4775.pdf

Compiler Support

- Implemented in MSVC
  - VS2015 (or VS2013 w/ Nov. ‘13 CTP) – pre-TS implementation
  - VS2017 – TS compliant (e.g. adding `co_` prefix)
  - Just add `/await` to the compilation flags
- Implemented in Clang
  - Since clang 5
  - Use `-fcoroutines-ts` (soon: `-std=c++2a`) `-stdlib=libc++`
- gcc implementation started
  - https://gcc.gnu.org/wiki/cxx-coroutines

Maturity

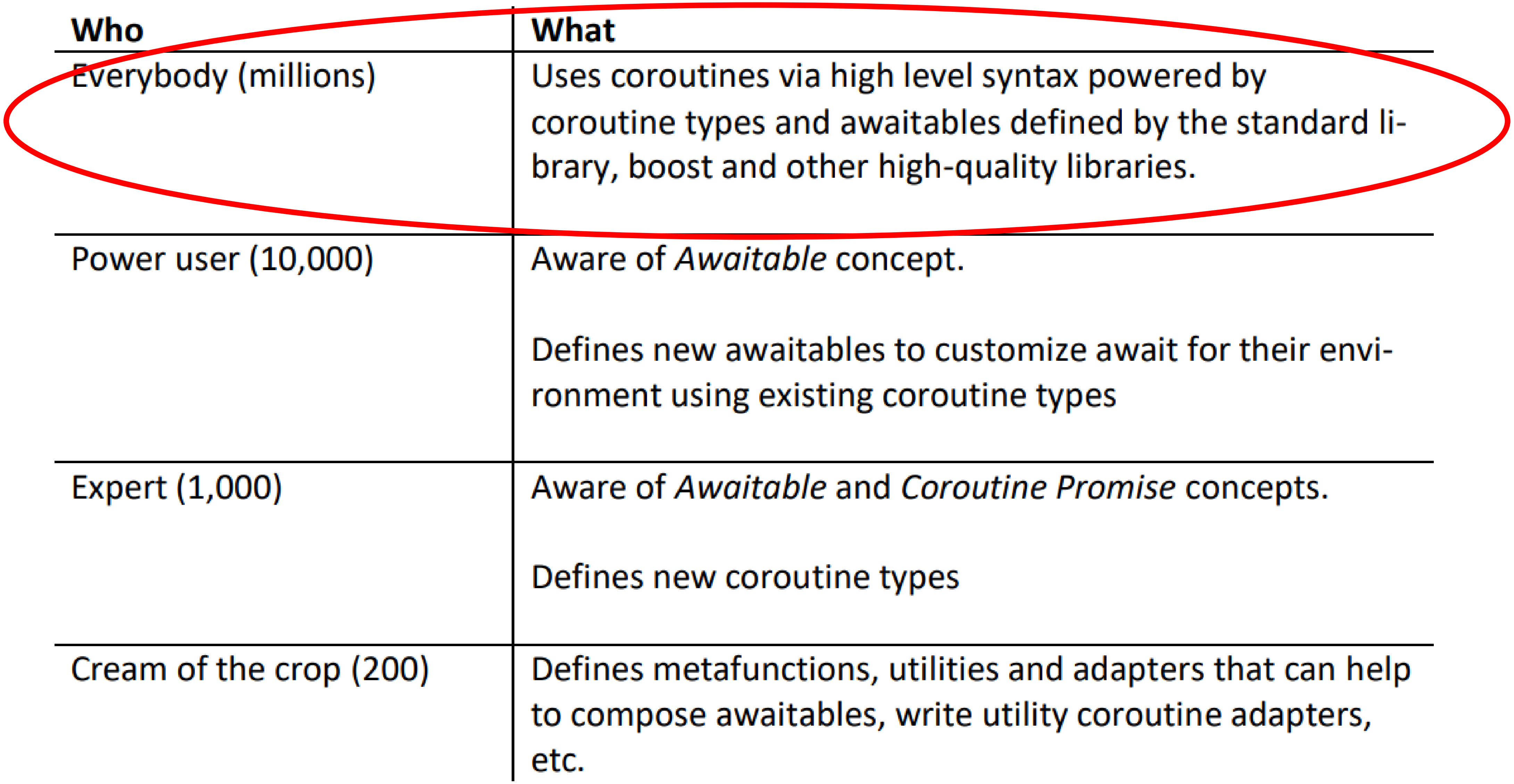


- 2012 – 2013: Early design, simple implementation, handles only a few of the use cases
- 2014: Complete design handling all use cases. Implementation shipped in an official release of the MSVC. We start accumulating feedback from C++ developers across the world.
- 2015: Feature freeze. No substantial changes afterwards. Implemented in EDG frontend. Production use in MSVC compiler, Prototype in Clang compiler.
- 2016: Production deployment on Linux using Clang compiler (using a public fork)
- 2017: Coroutines are in a Clang 5.0. TS is published.
- 2018: GCC implementation starts, first coroutine library types proposed for standardization

By the time we are in Kona (2019), the Coroutines TS would have accumulated:

- 5 years of feedback of C++ developers of using the Coroutines TS (4 years in its current form)
- 4 years of production deployment on Windows in business-critical software
- 3 years of production deployment on Linux in business-critical software
- Implementation experience in 4 major compilers: MSVC, Clang, EDG and GCC

https://wg21.link/p1362r0 by Gor Nishanov



Who	What
Everybody (millions)	Uses coroutines via high level syntax powered by coroutine types and awaitables defined by the standard library, boost and other high-quality libraries.
Power user (10,000)	Aware of *Awaitable* concept. Defines new awaitables to customize await for their environment using existing coroutine types
Expert (1,000)	Aware of *Awaitable* and *Coroutine Promise* concepts. Defines new coroutine types
Cream of the crop (200)	Defines metafunctions, utilities and adapters that can help to compose awaitables, write utility coroutine adapters, etc.

4.4 Are coroutines expert only feature?
https://wg21.link/p1362r0 by Gor Nishanov