Coroutines: More Details

This lesson clarifies more details regarding coroutines, including use-cases, design goals, and underlying concepts.

WE'LL COVER THE FOLLOWING

Typical Use-Cases

Coroutines are the natural way to write event-driven applications; e.g. simulations, games, servers, user interfaces, or even algorithms. Coroutines are typically used for cooperative multitasking. The key to cooperative multitasking is that each task takes as much time as it needs. This is in contrast to pre-emptive multitasking, for which we have a scheduler that decides how long each task gets the CPU.

That being said, there are different kinds of coroutines.

Underlying Concepts

Coroutines in C++20 are asymmetric, first-class, and stackless.

The workflow of an **asymmetric** coroutine goes back to the caller. This will not hold for a symmetric coroutine. A symmetric coroutine can delegate its workflow to another coroutine.

First-class coroutines are similar to First-Class Functions since coroutines behave like data. This means that you can use them as an argument to return

value from a function, or store them in a variable.

A *stackless* coroutine enables it to suspend and resume the top-level coroutine, but this coroutine can not invoke another coroutine.

Proposal N4402 describes the design goals of coroutines.

Design Goals

Coroutines should

- be highly scalable (to billions of concurrent coroutines).
- have highly efficient resume and suspend operations comparable in cost to the overhead of a function.
- seamlessly interact with existing facilities with no overhead.
- have open ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics such as generators, goroutines, tasks and more.
- usable in environments where exceptions are forbidden or not available.

There are four reasons for a function to become a coroutine.

Becoming a Coroutine

A function will become a coroutine if it uses,

- `co_return`, or
- `co_await`, or
- `co_yield`, or a
- `co_await` expression in a range-based for-loop.

This explanation was from proposal N4628.

Finally, I will discuss the new keywords `co_return`, `co_yield`, and `co_await`.

`co_return`, `co_yield`, and `co_await`

co_return: a coroutine uses **co_return** as its return statement.

co_yield: thanks to `co_yield` you can implement a generator. This means you can create a generator that will generate an infinite data stream from which you can successively query values. The return type of the generator `generator<int> generatorForNumbers(int begin, int inc= 1)` is `generator<int>`. `generator<int>` internally holds a special promise p such that a call `co_yield i` is equivalent to a call `co_await p.yield_value(i)`. `co_yield i` can be called an arbitrary number of times. Immediately after the call, the execution of the coroutine will be suspended.

co_await: `co_await` eventually causes the execution of the coroutine to be suspended and resumed. The expression `exp` in `co_await exp` has to be a so-called awaitable expression, and `exp` has to implement a specific interface. This interface consists of the three functions: `e.await_ready`, `e.await_suspend`, and `e.await_resume`.

The typical use case for `co_await` is a server that waits for events.

```
Acceptor acceptor{443};
while (true){
  Socket socket= acceptor.accept();              // blocking
  auto request= socket.read();                   // blocking
  auto response= handleRequest(request);
  socket.write(response);                        // blocking
}
```

The server is quite simple because it sequentially answers each request in the same thread. The server listens on port 443 (line 1), accepts its connections (line 3), reads the incoming data from the client (line 4), and writes its answer to the client (line 6). All calls in lines 3, 4, and 6 are blocking.

Thanks to co_await, the blocking calls can now be suspended and resumed.

```
Acceptor acceptor{443};
while (true){
  Socket socket= co_await acceptor.accept();
  auto request= co_await socket.read();
  auto response= handleRequest(request);
  co_await socket.write(response);
}
```