

Coroutines

Stephen W. Carson
Refinitiv

	Execution State	Thread of Execution
Thread	1	1
Coroutine	1	0

Think Cooperative Multitasking!

- Coroutine:
 - Gets the thread of execution
 - Does some work
 - Yields the thread of execution
 - (Remembers execution location + all “stack” variables)

“Coroutines, like if, loops and function calls, are another kind of “structured goto” that lets you express certain useful patterns (like state machines) in a more natural way.”

Coroutines in C++

	Subroutine	Coroutine
Invoke	Function call, e.g. f()	Function call, e.g. f()
Return	<code>return</code> statement	<code>co_return</code> statement
Suspend		<code>co_await</code> expression <code>co_yield</code> expression (suspend and return value)
Resume		<code>coroutine_handle<>::resume()</code>
Destroy		<code>coroutine_handle<>::destroy()</code>

```
generator<int> get_integers( int start=0, int step=1 ) {  
    for (int current=start; true; current+= step)  
        co_yield current;  
}
```

```
Generator<int> getNext(int start = 0, int step = 1){  
    auto value = start;  
    for (int i = 0;; ++i){  
        co_yield value;           // 1  
        value += step;  
    }  
}
```

```
int main() {  
  
    std::cout << std::endl;  
  
    std::cout << "getNext():";  
    auto gen = getNext();  
    for (int i = 0; i <= 10; ++i) {  
        gen.next();               // 2  
        std::cout << " " << gen.getValue();  
    }  
}
```



```
std::future<std::expected<std::string>> load_data( std::string resource )
{
    auto handle = co_await open_resource(resource);
    while( auto line = co_await read_line(handle)) {
        if (std::optional<std::string> r = parse_data_from_line( line ))
            co_return *r;
    }
    co_return std::unexpected( resource_lacks_data(resource) );
}
```

Coroutine Mechanics

C++ Coroutines Can't Use

- variadic arguments
- plain return statements
- placeholder return types (`auto` or Concept)

C++ Coroutines Can't Be

- Constexpr functions
- constructors
- destructors
- main function

promise

- A promise object is manipulated inside the coroutine.
- The coroutine submits its result or exception through this object.
- You needn't deal with this promise directly! It is mostly under the hood.

Coroutine handle

- Refers to the coroutine's dynamically allocated state
- This is used outside of the coroutine to:
 - resume execution of the coroutine
 - or destroy the coroutine frame

Coroutine State

- Heap-allocated (unless the allocation is optimized out)
- Contains:
 - the promise object
 - the parameters (all copied by value)
 - some representation of the current suspension point, so that resume knows where to continue and destroy knows what local variables were in scope
 - local variables and temporaries whose lifetime spans the current suspension point

Co_awaiting Coroutines

co_await operator

- Forces compiler to generate some coroutine boilerplate code
- Creates the *Awaiter* object

Simplest Awaitables

- `suspend_always`
 - `co_await suspend_always` will always suspend the coroutine and return back to the caller
- `suspend_never`
 - `co_await suspend_never` will never suspend the coroutine

Library Awaitables

cppcoro awaitables

- `single_consumer_event`
- `single_consumer_async_auto_reset_event`
- `async_mutex`
- `async_manual_reset_event`
- `async_auto_reset_event`
- `async_latch`
- `sequence_barrier`
- `multi_producer_sequencer`
- `single_producer_sequencer`

single_consumer_event

```
#include <cppcoro/single_consumer_event.hpp>

cppcoro::single_consumer_event event;
std::string value;

cppcoro::task<> consumer()
{
    // Coroutine will suspend here until some thread calls event.set()
    // eg. inside the producer() function below.
    co_await event;

    std::cout << value << std::endl;
}

void producer()
{
    value = "foo";

    // This will resume the consumer() coroutine inside the call to set()
    // if it is currently suspended.
    event.set();
}
```

async_mutex

```
#include <cppcoro/async_mutex.hpp>
#include <cppcoro/task.hpp>
#include <set>
#include <string>

cppcoro::async_mutex mutex;
std::set<std::string> values;

cppcoro::task<> add_item(std::string value)
{
    cppcoro::async_mutex_lock lock = co_await mutex.scoped_lock_async();
    values.insert(std::move(value));
}
```

Learn More

- [GitHub - lewissbaker/cppcoro: A library of C++ coroutine abstractions for the coroutines TS](#)
- [What are use-cases for a coroutine? - Stack Overflow](#)
- [Iterator - Generators - Wikipedia](#)
- [Coroutines \(C++20\) - cppreference.com](#)
- [c++ - What are coroutines in C++20? - Stack Overflow](#)
- [When should I use coroutines? : Unity3D](#)

- [CppCon 2015: Gor Nishanov “C++ Coroutines - a negative overhead abstraction” - YouTube](#)
- [CppCon 2016: James McNellis “Introduction to C++ Coroutines” - YouTube](#)
- [GitHub - lewissbaker/cppcoro: A library of C++ coroutine abstractions for the coroutines TS](#)
- [C++ links: Coroutines \(WIP draft\) · GitHub](#)

Coroutines in Python

- [David Beazley - Python Concurrency From the Ground Up: LIVE! - PyCon 2015 - YouTube](#)
- [Curious Course on Coroutines and Concurrency - YouTube](#)
- [A Curious Course on Coroutines and Concurrency](#)
- Also see Ch. 16 in [Fluent Python: Clear, Concise, and Effective Programming: Luciano Ramalho: 4708364244547: Amazon.com: Books](#)
 - Available on O'Reilly Books Online