Coroutines TS for C++2a

Charlie Kolb

3rd July 2018

Unless specified otherwise, all information regarding the specification of coroutines are taken from the Coroutines Technical Specification. [1]

1 Introduction

A coroutine is a resumable function. As opposed to subroutines - the currently available type of function in C++ -, coroutines are not only callable, and able to return a value, but are also able to *suspend* execution and be resumed at a later stage.

This enables *asynchronous* execution of a single function without a complete change in code style.

The Coroutines Technical Specification proposes the addition of Coroutines and related keywords and std utilities for a unified interface, allowing for both more concise code, and a well defined framework for optimizations to be applied in. The proposed features enable a new way of control flow, making this TS very interesting for various things not necessarily related to asynchronous execution.

Do note that Coroutines are not yet accepted into the C++20 standard, and thus are neither guaranteed to be added to the standard, nor to be added in the precise form. That said, the proposal has the backing of major voices in the C++ Community like Bjarne Stroustrup [2] and is already mostly implemented in clang (5.0 onwards) and VC++.

We will consider the syntax and use cases of coroutines and in particular focus on the difference between user experience and the caveats a library writer supporting coroutines faces.

2 Patch Notes

The Technical Specification defines three new keywords and a new loop construct:

- co_yield
- co_await
- co_return
- for co_await loop

Additionally, their following structs are to be added to the std::experimental header[1]:

- suspend_always
- suspend_never
- coroutine_handle<T>
- coroutine_traits<T>

3 Usage

To explain the concepts of coroutines, we will assume the existence of a struct "return_t", which provides certain functions which fulfill the requirement set by the operators. We will examine these requirements in a later chapter.

3.1 Concurrency by example

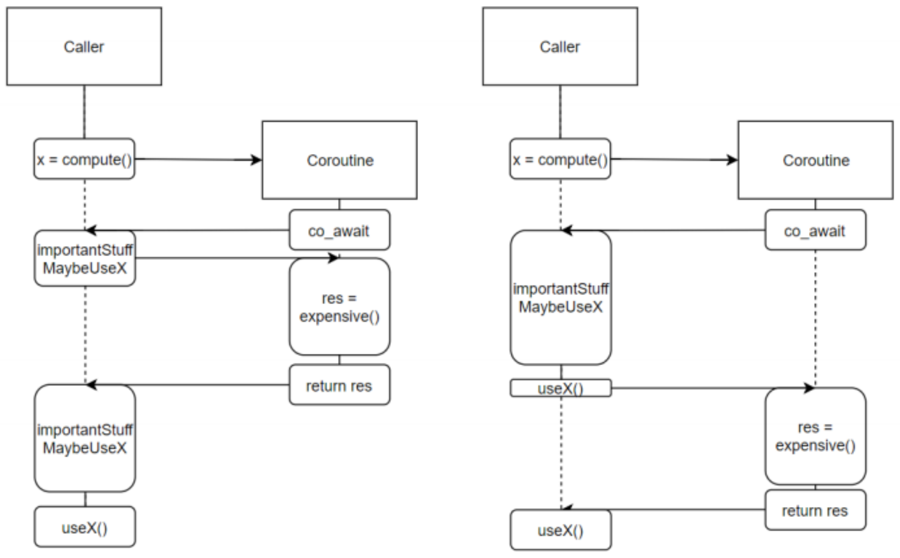
Consider the following program

[1]It seems unlikely for the structs to stay in experimental, but there is no mention of another place for them in a final release. A reasonable guess would be incorporate them into a new <coroutine> header (Which currently exists in std::experimental)

```
return_t compute() {
    int result = abs(
        co_await std::bind(someExpensiveFn)
    );
    co_return 5 * result * result + 2 * result;
}

int main() {
    auto x = compute(); // call of the coroutine
    // rest of the program, may or may not use x
    useX(x);
}
```

Here, *compute* is a coroutine which may suspend execution before calling *someExpensiveFn* thanks to the co_await operator. The coroutine is resumed only once the actual return value is needed. (Depending on your coroutine return type this may happen in an implicit cast to the real return value or with a blocking function call like future::get()).

This way, we have two possibilities of control flow, determined by whether "importantStuffMaybeUseX" uses x:



Now we only compute `someExpensiveFn` once it is necessary.

Note that this program runs *concurrently*, but not in parallel. While coroutines enable easy parallelization, they are by themselves are a seperate concept.

Of course, with futures, promises and async, there already exists a solution for concurrency in the standard library. These however require great changes in syntax when compare to the synchronous versions, and are hard to incorporate into longer functions which may for example continue and then suspend again before the final result is returned.

If we were to consider a synchronous version of compute, we only need to change the return type to int, remove the co_await operator, exchange co_return with return and call *someExpensiveFn* directly.

Admittedly, coroutines aren't a magical solution which enables these types of changes. In the background, we need many things to support this example, like certain functions and operator overloads in order to make it look this normal. In practical use cases, these responsibilities all lie on the side of a library which provides you with appropriate types, hiding the complexity from the end user. We will take a look at these requirements in a later chapter.

3.2 Generator with co_yield

A generator is a structure which produces values on demand, without the cost of computing them upfront. This allows for low memory requirements and may in theory go on forever, without any direct cost.

The co_yield keyword allows for suspension of the coroutine, and may return a value at the same type.

```
generator_t fibGen() {
    int a = 0;
    int b = 1;
    for(int i = 0;;i++) {
        co_yield b;
        int temp = b;
        b += a;
        a = b;
    }
}

int main() {
    auto fibbage = fibGen();

    fibbage.next(2); // 1 1
    fibbage.next(5); // 2 3 5 8 13
}
```

This custom generator type provides utility functions to access the next elements, but it is also possible to incorporate this into a range-based for loop.

3.3 for co_await

The new for co_await loop is a range based loop which co_await on every element in the container **before** the loop-iteration is run.

3.4 Suspension

Whether a coroutine suspends depends on the argument of the co_await or co_yield operator. The standard library provides struct which always, or never suspend:

```
return_t coro() {
    std::cout << "First call!\n";
    co_await std::experimental::suspend_never{};
    std::cout << "Still first call!\n";
    co_await std::experimental::suspend_always{};
    std::cout << "Only after resumed!\n"
}
```

Both the coroutine in general and co_await operator require the existence of certain functions.

4 Behind the scenes

4.1 co_await

To see what happens under the hood, let's look at what happens when the compiler encounters a statement like

```
auto result = co_await <expr>;
```

If we were to achieve the same thing as this statement, it would look somewhat like this [3]:

```
auto&& __a = <expr>;
if (!__a.await_ready())
{
    if(returnType(__a.await_suspend) == void  ||
       (returnType(__a.await_suspend) == bool &&
         __a.await_suspend(coroutine-handle))
        // suspend execution
}
auto result = __a.await_resume();
```

The co_await operator requires three functions to be callable on our expr:

- bool await_ready(): Checks whether we even should suspend
- (*void|bool|coroutine_handle*)[2] await_suspend(coroutine_handle): May decide not to suspend or prepare other constructs **before** the suspension, meaning the argument itself is executed, and may need to be wrapped in another function which is asynchronous (i.e. a coroutine with only a co_return statement) before control is suspended to the caller.
- auto await_resume(): Returns the actual output of the co_await statement, i.e. our result will hold this function's return value

If one wants to input a type, which does not provide these functions, as argument for the co_await operator, one must overload the co_await operator for that argument's type and return an object which does provide these(i.e. return suspend_always).

This design allows for insane use-cases, like using the co_await operator on functions which return a std::optional, and use the above mechanism to suspend permanently if the optional is empty, returning an empty optional to the original caller, or continue execution (and optionally unwrap the optional with await_resume()) in the coroutine, enabling optionals to work entirely without annoying, manual checks.

The coroutine-handle required by the await_suspend function is provided by the coroutine which contains the co_await operator and acts like a this-pointer for this instance of the function.

For a coroutine using no co_return operator or one without an argument, the handle offers the following interface:

```
template <>
struct coroutine_handle<void>
{
coroutine_handle() noexcept = default;
coroutine_handle(nullptr_t) noexcept;
coroutine_handle& operator=(nullptr_t) noexcept;
explicit operator bool() const noexcept;

// c style interfacing
static coroutine_handle from_address(void* a) noexcept;
void* to_address() const noexcept;
```

[2]Exactly one await_suspend function must be provided, either with void, bool, or another coroutine_handle as return type. In the last case, the returned handle will automatically be resumed (This may continue recursively), and afterwards this co_await call is suspended

```
// control of the coroutine execution
void operator()() const;
void resume() const;
void destroy();
bool done() const;
};
```

which is a specialization for the general templated struct used for a return type of templated type *Promise*, which provides two additional functions:

```
template <typename Promise>
struct coroutine_handle : coroutine_handle<void>
{
    Promise& promise() const noexcept;
    static coroutine_handle from_promise(Promise&) noexcept;
};
```

4.2 co_yield

A statement like

```
co_yield 5;
```

is simply transformed into

```
co_await context->_promise.yield_value(5);
```

which exposes the context with a field called _promise. This is part of the hidden *state* of a coroutine.

4.3 Coroutine States

As a coroutine ought to "remember" where it left of, it requires a local storage. This storage is called the *coroutine_state*. It holds the coroutine_handle, local and captured variables[3] and the execution state of the function (i.e. registers, instruction pointer)[4].

In *general*, dynamic allocation is required[5] for the creation of this context object. As the interfaces are very well defined, functions and types used will share much resemblance. This combined with the fact that all scheduling is automatically done at compile-time makes this dynamic allocation and coroutines in general a prime target for optimization.[6]

[3]with capture rules very closely alligned with the lambda capture rules

[4]This is called a *stackless* coroutine, which, unlike to the *stackful* coroutine does not capture the entire stack.

[5]One may circumvent this by overloading the operator new, mentioned in the following chapter

[6]See [4] for a nice example

4.4 Coroutine expanded

The hidden parts of a coroutine are shown by the following example:

```
return_t fn() {
    co_await suspend_always{};
    co_return 20;
}
```

expands to

```
return_t fn() {
    // before the start of the function body
    _fn_context* _context = new _fn_context{};

    // _return will be returned the first
    // time we suspend execution
    _return = _context->_promise.get_return_object();
    co_await _context->_promise.initial_suspend();

    // start of the function body
    co_await suspend_always{};

    // co_return becomes...
    _context->_promise.return_value(20);
    goto _final_suspend_label;

    // end of the normal function body
    co_await _context->_promise.final_suspend();
    delete _context;
}
```

As we can see, our return type may choose to immediately suspend execution before entering the actual function body, implying a lazy execution model.

Note that, should the return type not suspend in final_suspend(), the context will be deleted, and with it the promise object which will hold our potential return value, implying the return object may be deleted by the time it is used if the coroutine is not suspended in final_suspend. [7]

The context->_promise is defined by our return_t, which thus needs to implement the following functions:

[7]In the suspended case the context ought to be deleted manually with a final resume() call or a direct call to coroutine_handle¡¿::destroy(), preferably in return_t's destructor to ensure no memory is leaked.

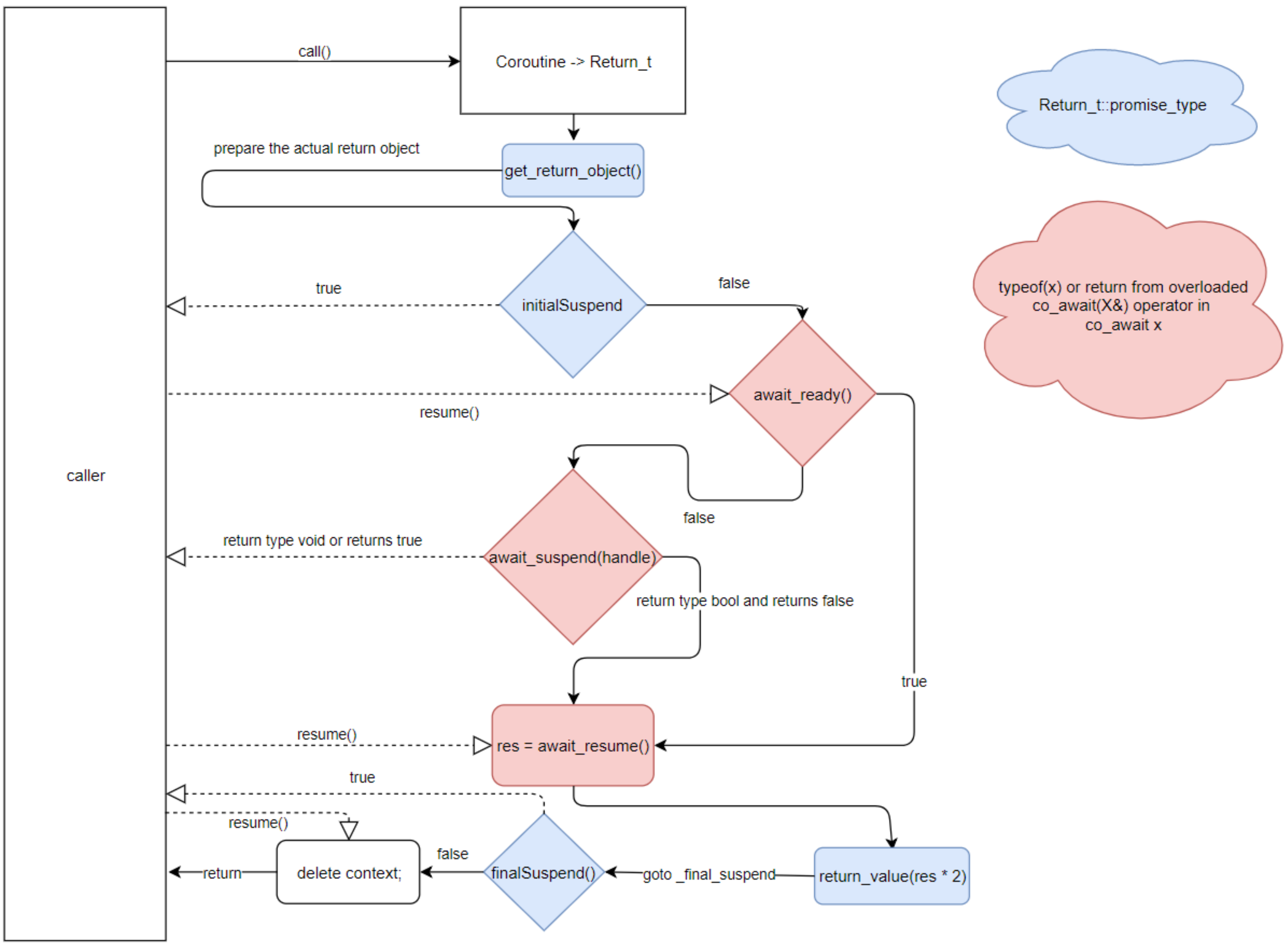
```
struct return_t {
    struct promise_type {
        // Fallback if dynamic allocation failed
        static auto get_return_object_on_allocation_failure();

        // The result of this function is the
        // actual return value of the coroutine call
        // body as example
        auto get_return_object() {
            return return_t(
                coroutine_handle<promise_type>
                    ::from_promise(*this);
            )
        };

        // Used to choose whether to suspend
        // Return object will be argument for co_await
        // So they need the three await_X functions!
        auto initial_suspend();
        auto final_suspend();

        // invoked if the coroutine body threw an exception
        void unhandled_exception();

        // Only one of these
        auto return_value(X x); // called with 'co_return x;'
        void return_void();     // called with 'co_return ;'

        // Only required if co_yield is used
        // return type needs to implement the await_x functions
        auto yield_value(int v);
    };
};
```

4.5 Branch Options visualized

To visualize the options a coroutine presents us, consider the control flow options for the following coroutine:

```
Return_t coro() {
    auto res = co_await <expr>;
    co_return res;
}
```

.



The blue parts are controlled by the return_t::promise_type function, the red parts by the await_X functions provided either directly through ¡expr¿ or through an overloaded co_await ¡expr¿ operator. A dotted line implies suspension of the coroutine if going towards the calller and resumption if coming from the caller.

4.6 Shortcuts

If we do not want to implement all these functions directly in our return type (for either clarity issues or for things we can't or aren't supposed to modify like standard library classes), we may use the std::experimental::coroutine_traits struct, i.e. to implement coroutine support for std::future [3]:

```
template<typename T, typename... Args>
struct coroutine_traits<return_t<T>,Args...> {
    struct promise_type {
        // Our actual storage unit, not required per specification
        // but we want to store our later return object somehow
        promise<T> _promise;

        future<T> get_return_object() {
            return _promise.get_future();
        }

        auto initial_suspend() { return suspend_never{}; }
        auto final_suspend() { return suspend_always{}; }

        template<typenme U>
        void return_value(U&& v) {
            _promise.set_value(std::forward<U>(v));
        }
    }
}
```

This allows us to use future as return type, to use it as co_await argument, we need to implement a wrapper for the await functions and overload co_await [3].

```
template<typename T>
struct future_awaiter {
    future<T>& _f;

    bool await_ready() { return _f.is_ready(); }
    void await_suspend(coroutine_handle<> ch) {
        _f.then([ch]() { ch.resume(); });
    }
    auto await_resume() { return _f.get(); }
};

template <typename T>
future_awaiter<T> operator co_await(future<T>& value) {
    return future_awaiter<T>{value};
}
```

Do note that only the C++20 version of future, currently available as std::experimental::future contains .is_ready and .then functions.

5 Evaluation

Coroutines offer basic building blocks for asynchronous programs and libraries. As they are baked into the language with keywords, they will provide a standardized way of doing asynchronous programming.

Most of the time, coroutines come at an insanely low cost, enabled by the low overhead and compile-time scheduling and the resulting possible optimizations. It is possible to have millions of coroutines running on a modern pc, enabling tasks like priority based scheduling.

Additionally, coroutines offer us a new way of control flow, which keeps the code very clean (lack of indention) and compact.

That said, in particular the hidden control flow shows us some of the dangers of these techniques. The control flow which determines execution is hidden, and it might not immediately be obvious what a coroutine is doing just by looking at it.

For example, a co_await on a network connection may either suspend if it was not possible to create a connection, or may suspend with the intention of creating a connection the next time from the start. While the end result is the same, one of these options will block the execution, with no difference apparent without consulting the connection function source code or documentation.

While the dynamic allocation is often optimizable, it is hard to fix when it isn't. In particular when libraries provide the used return types, it may not be an option to change the libraries behaviour for the particular use case.

The naming of the operators is also questionable, co_await may choose to not suspend, co_yield may choose to not yield. The names also imply the use cases being tied to asynchronous functions, disregarding use cases like indention-free work with optionals.

5.1 Use cases

Coroutines will find use in general asynchronous programming, replacing library level features with language level constructs. Generators are a new tool for our software, in particular for those who require lazy evaluation. Scheduling is another big use case here, one may easily implement a real time system where tasks are worked on and only suspend once a certain time limit comes close. The last big use case is UI design, where operating system calls may be suspended on, waiting for your OS to respond and still providing a smooth user experience in the meantime.

References

[1] G. Nishanov. (2018) Coroutines ts. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4760.pdf

[2] B. Stroustrup. (2018) Remember the vasa. [Online]. Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0977r0.pdf

[3] J. McNellis. (2016) Cppcon 2016: Introduction to c++ coroutines. [Online]. Available: https://www.youtube.com/watch?v=ZTqHjjm86Bw

[4] Example for coroutine optimization potential. [Online]. Available: godbolt.org/g/26viuZ