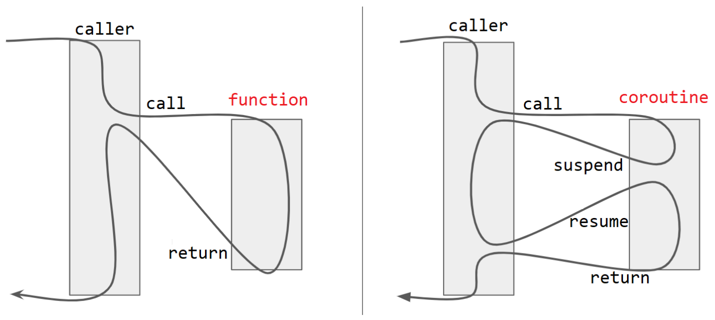
C++20 Coroutine: Under The Hood



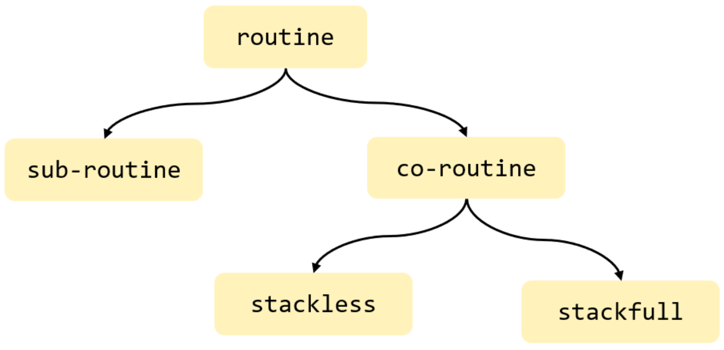
Reading Time: 12 minutes

A coroutine is one of the major feature introduced with the C++20 standard apart from Module, Ranges & Concept. And you see how happy I am to unfold it. I already set the baseline on this topic with my previous article that Coroutine in C Language, where we saw, how suspension-resumption of execution works! With this article "C++20 Coroutine: Under The Hood", we will see how compiler creates magic & standard library helps it with basic infrastructure making C++20 coroutine more sophisticated(yet complex) & scalable/customizable.

At any point, you feel there is some jargons/terminology that is not known to you. Please keep reading forward. I have added a special section for it.

Contents [hide]

What Is Coroutine In General?



- Please refer my previous article Coroutine in C Language for more.

What Is C++20 Coroutine?

- A function that
  1. Contains keywords `co_await`, `co_yield` and/or `co_return`.
  2. Use a return type specifying a promise.
- From the higher abstraction, the C++20 coroutine consists of:
  1. Promise

        - Defines overall coroutine behaviour.
        - Act as a communicator between caller & called coroutine.

2. Awaiter

    - Controls suspension & resumption behaviour.

3. Coroutine handle

    - Controls execution behaviour.

Why Do You Even Need Coroutine?

- I have already covered this in my earlier post Coroutine in C Language.
- However, if you still want to understand the need for coroutine with use case then please refer to Iterator Design Pattern With Modern C++.
- Or you can directly see the section of this article Generating Integer Sequence Using Coroutine.

Understanding C++20 Coroutine With Examples

- Enough theory. Let's talk code.

Suspending a Coroutine

- Following is a lame & minimal example of a C++20 Coroutine. However, this is a very good starting point for beginner with slow pace & less cluttered code.

```
#include <coroutine>
#include <iostream>

struct HelloWorldCoro {
    struct promise_type { // compiler looks for `promise_type`
        HelloWorldCoro get_return_object() { return this; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }
    };

    HelloWorldCoro(promise_type* p) : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~HelloWorldCoro() { m_handle.destroy(); }

    std::coroutine_handle<promise_type>      m_handle;
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await std::suspend_always{};
    std::cout << "World!" << std::endl;
}

int main() {
    HelloWorldCoro mycoro = print_hello_world();

    mycoro.m_handle.resume();
    mycoro.m_handle(); // Equal to mycoro.m_handle.resume();
    return EXIT_SUCCESS;
}
// g++-10 -std=c++20 -fcoroutines -fno-exceptions -o myapp Main.cpp
```

- A coroutine can be resumed by a resume member function of the std::coroutine_handle or by invoking the function call operator of the std::coroutine_handle object.
- As I have mentioned earlier, the C++20 coroutine consists of:

    1. Promise i.e. promise_type

        - Type containing special methods like get_return_object(), initial_suspend(), final_suspend(), etc. that compiler use in coroutine transformation. Hence, it controls overall coroutine behaviour.

    2. Awaiter i.e. std::suspend_always

        1. Empty class to suspend coroutine always.

        - You can see the GCC implementation for this type here.

    3. Coroutine handle i.e. std::coroutine_handle

        - Handler for coroutine that used to resume, destroy or check on a lifetime of coroutine.

- The compiler transforms the above print_hello_world coroutine as :

```
HelloWorldCoro print_hello_world() {
    __HelloWorldCoro_ctx* __context = new __HelloWorldCoro_ctx{};
    auto __return = __context->_promise.get_return_object();
    co_await __context->_promise.initial_suspend();

    std::cout << "Hello ";
    co_await std::suspend_always{ };
    std::cout << "World!" << std::endl;

__final_suspend_label:
    co_await __context->_promise.final_suspend();
    delete __context;
    return __return;
}
```

- As you can see, the compiler first creates the context(i.e. coroutine state) object. Which presumably look like:

```
struct __HelloWorldCoro_ctx {
    HelloWorldCoro::promise_type _promise;
    // storage for argument passed to coroutine
    // storage for local variables
    // storage for representation of the current suspension point
};

// Standard doesn't define such type, rather compilers choose the type that suits its implementation.
```

- As seen, with the help of this context object, it creates the HelloWorldCoro object named as __return with the help of promise's method get_return_object().
- Finally, one more level of transformation of co_await statements, coroutine would look like:

```
HelloWorldCoro print_hello_world() {
    __HelloWorldCoro_ctx* __context = new __HelloWorldCoro_ctx{};
    auto __return = __context->_promise.get_return_object();
    {
        auto&& awaiter = std::suspend_always{};
        if (!awaiter.await_ready()) {
            awaiter.await_suspend(std::coroutine_handle<> p);
            // compiler added suspend/resume hook
        }
        awaiter.await_resume();
    }

    std::cout << "Hello ";
    {
        auto&& awaiter = std::suspend_always{};
        if (!awaiter.await_ready()) {
            awaiter.await_suspend(std::coroutine_handle<> p);
            // compiler added suspend/resume hook
        }
        awaiter.await_resume();
    }
    std::cout << "World!" << std::endl;

__final_suspend_label:
    {
        auto&& awaiter = std::suspend_always{};
        if (!awaiter.await_ready()) {
            awaiter.await_suspend(std::coroutine_handle<> p);
            // compiler added suspend/resume hook
        }
        awaiter.await_resume();
    }
    return __return;
}
```

- As you can see our `HelloWorldCoro::promise_type::initial_suspend()` & `HelloWorldCoro::promise_type::final_suspend()` returns `std::suspend_always`.
- Whose, method `std::suspend_always::await_ready()` in turn returns `false` always.
- So our coroutine will be suspended every time it encounters `co_await`.

Returning a Value From Coroutine

```
#include <coroutine>
#include <iostream>
#include <cassert>

struct HelloWorldCoro {
    struct promise_type {
        int m_value;

        HelloWorldCoro get_return_object() { return this; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }

        void return_value(int val) { m_value = val; }
    };

    HelloWorldCoro(promise_type* p) : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~HelloWorldCoro() { m_handle.destroy(); }

    std::coroutine_handle<promise_type>      m_handle;
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await std::suspend_always{ };
    std::cout << "World!" << std::endl;

    co_return -1;
}
```

```
int main() {
    HelloWorldCoro mycoro = print_hello_world();

    mycoro.m_handle.resume();
    mycoro.m_handle.resume();
    assert(mycoro.m_handle.promise().m_value == -1);
    return EXIT_SUCCESS;
}
```

- To return a value from a coroutine, you need to supply `return_value()` method to promise type. In other words, the compiler expects the method named `return_value` with appropriate argument.
- And, if you don't supply it to promise type, you will be prompted with the below error:

```
Main.cpp:27:5: error: no member named 'return_value' in 'std::__n4861::__coroutine_traits_impl<HelloWorl
    27 |     co_return -1;
       |     ^~~~~~~~~
```

- This `return_value()` method is then used by compiler to transform the `co_return` statement as below:

```
HelloWorldCoro print_hello_world() {
    __HelloWorldCoro_ctx* __context = new __HelloWorldCoro_ctx{};
    auto __return = __context->_promise.get_return_object();
    co_await __context->_promise.initial_suspend();

    std::cout << "Hello ";
    co_await std::suspend_always{ };
    std::cout << "World!" << std::endl;

    __context->_promise.return_value(-1);
    goto __final_suspend_label;

__final_suspend_label:
    co_await __context->_promise.final_suspend();
    delete __context;
    return __return;
}
```

- As you might have noticed, `co_return` transformation is plain vanilla, where the compiler just passes the return value to the promise object & jump to the final suspend label.
- I will not expand the `co_await` statements again. Otherwise, the code will become messy. However, by now, you can guess what could be there.

Yielding a Value From Coroutine

```
#include <coroutine>
#include <iostream>
#include <cassert>

struct HelloWorldCoro {
    struct promise_type {
        int m_val;

        HelloWorldCoro get_return_object() { return this; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }

        std::suspend_always yield_value(int val) {
            m_val = val;
```

```
            return {};
        }
    };

    HelloWorldCoro(promise_type* p) : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~HelloWorldCoro() { m_handle.destroy(); }

    std::coroutine_handle<promise_type>      m_handle;
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_yield 1;
    std::cout << "World!" << std::endl;
}

int main() {
    HelloWorldCoro mycoro = print_hello_world();

    mycoro.m_handle.resume();
    assert(mycoro.m_handle.promise().m_val == 1);
    mycoro.m_handle.resume();
    return EXIT_SUCCESS;
}
```

- As we have done in returning a value from a coroutine, to yield anything from coroutine you need to supply the `yield_value()` method to promise_type that returns awaitable type.
- Once again, the compiler transforms the coroutine that yields as:

```
HelloWorldCoro print_hello_world() {
    __HelloWorldCoro_ctx* __context = new __HelloWorldCoro_ctx{};
    auto __return = __context->_promise.get_return_object();
    co_await __context->_promise.initial_suspend();

    std::cout << "Hello ";
    co_await __context->_promise.yield_value(1);
    std::cout << "World!" << std::endl;

__final_suspend_label:
    co_await __context->_promise.final_suspend();
    delete __context;
    return __return;
}
```

- As you can see `co_yield` statement transformed into `co_await` considering the promise's method(i.e. `yield_value()`) call as expression. That returns `std::suspend_always` so our coroutine will be suspended thereafter yielding a value.

Generating Integer Sequence Using C++20 Coroutine

- So, that we understood the compiler transformations of coroutine, let's do something meaningful with coroutine.

```
#include <coroutine>
#include <iostream>
#include <cassert>

struct Generator {
    struct promise_type {
        int m_val;
        Generator get_return_object() { return this; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }

        std::suspend_always yield_value(int val) {
            m_val = val;
            return {};
        }
    };

    /* ---------------------------- Iterator Implementation ---------------------------- */
    struct iterator {
        bool operator!=(const iterator& rhs) { return not m_h_ptr->done(); }
        iterator& operator++() {
            m_h_ptr->resume();
            return *this;
        }
        int operator*() { return m_h_ptr->promise().m_val; }

        std::coroutine_handle<promise_type> *m_h_ptr;
    };

    iterator begin() { return iterator{&m_handle}; }
    iterator end() { return iterator{nullptr}; }
    /* ------------------------------------------------------------------------------------ */

    Generator(promise_type* p) : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~Generator() { m_handle.destroy(); }

    std::coroutine_handle<promise_type>      m_handle;
};


Generator range(uint32_t start, uint32_t end) {
    while(start != end)
        co_yield start++;
}

int main() {
    for (auto &&no : range(0, 10)) {  // Isn't this look like Python !
        std::cout<< no <<std::endl;
    }
    return EXIT_SUCCESS;
}
```

- A `range()` function of the above code will generate the integers on every iteration on a need basis. Instead of generating & returning precomputed sequence.
- For the rest of the code, I suggest you spend some time with patience. And play around it.
- And, If you are thinking that why do we need `iterator` implementation inside `Generator`, then please learn How Range Based `for` Loop Works In C++!.

C++20 Coroutine Terminology

- This section should have been at beginning of the article. However, I believe that until you understand how coroutine works under the hood, you won't be able to connect the dots on C++20 coroutine jargons(or won't understand data type naming convention).
- By this time you are able to write a C++ coroutine & you can ignore this section. Nonetheless, it is just that you will have a hard time communicating with fellow C++ developers & reading cppreference. If you do so.

Awaitable

- A type that supports the `co_await` operator is called an Awaitable type.
- C++20 introduced a new unary operator `co_await` that can be applied to an expression. For example:

```
struct dummy { // Awaitable
    std::suspend_always operator co_await(){ return {}; }
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await dummy{};
    std::cout << "World!" << std::endl;
}
```

Awaiter

```
struct my_awaiter {
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<>) {}
    void await_resume() {}
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await my_awaiter{};
    std::cout << "World!" << std::endl;
}
```

- An Awaiter type is a type that implements the three special methods that are called as part of a `co_await` expression: `await_ready()`, `await_suspend()` and `await_resume()`. For example, standard library defined trivial awaiters i.e. `std::suspend_always` & `std::suspend_never`.
- Note that a type can be both an Awaitable type and an Awaiter type.

co_await

- `co_await` is a unary operator that suspends a coroutine and returns control to the caller. Its operand is an expression whose type must either define operator `co_await`, or Awaitable.

```
struct dummy { }; // Awaitable

struct HelloWorldCoro {
    struct promise_type {
        // . . .
        auto await_transform(const dummy&) {
            return std::suspend_always{};
        }
    };
    // . . .
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await dummy{};
    std::cout << "World!" << std::endl;
}
```

Promise

- A type strictly named as *promise_type*.
- The promise object is used inside the coroutine. The coroutine submits its result or exception through this object.
- The promise type is determined by the compiler from the return type of the coroutine using std::coroutine_traits.

Coroutine Handle

- The coroutine handle used to resume the execution of the coroutine or to destroy the coroutine frame. It does also indicates the status of coroutine using `std::coroutine_handle::done()` method.

Coroutine State

- The coroutine state(referred to as a context object) is a compiler-generated, heap-allocated (unless the allocation is optimized out) object that contains
    - Promise object.
    - Parameters (all copied by value).
    - Local variables.
    - Representation of the current suspension point, so that resume knows where to continue and destroy knows what local variables were in scope.

How Does Coroutine Gets Executed!

- Ahh…! it's time for the real magic!
- Now, that we saw how the compiler does the magic with coroutine keywords `co_await`, `co_yield` & `co_return`. Let's understand how it gets executed.
- But, Before, we move forward, let me clarify some points:
    1. C++ Standard doesn't bother about implementation rather it only states behaviour. Hence implementation of the coroutine is completely dependent on the compiler & library writers. To see that, you can check the library implementation of `std::coroutine_handle::resume()` that uses __builtin_coro_resume which is provided by the GCC compiler. Proving that the library & compiler is tightly coupled.
    2. With that being said, I have not written any C++ library or compiler in my career. Though I have written/ported the core C library for proprietary architecture. So, whatever you see in this section is just to build your intuition on how coroutine can be implemented. Based on my research & intuition.
- There are two ways I can see this can be done:

Hypothesis 1

- For example, If you have defined your coroutine as:

```
HelloWorldCoro print_hello_world(int a) {
    int b = 10;
    co_await std::suspend_always{};
    a = 10;
    int c = 10;
    co_await std::suspend_always{};
    co_return a + b + c;
}
```

- Then, operations can be divided into 3 different sections separated by 2 suspension-point(i.e. `co_await <expr>`) as

  1. `int b = 10;`
  2. `a = 10; int c = 10;`
  3. `co_return a + b + c;`

- Considering these suspension points, then the compiler basically

  - Rewrite your coroutine into a bit like functions.

  - And create a copy of all local variables as a data member of the coroutine state object.

- You can assume coroutine state object layout as:

```
struct print_hello_world {

    print_hello_world(int a_) : a{a}{} // Coroutine state object with coroutine arguments

    void _s1(){ // Before suspension point 1 operations
        int b = 10;

        sp = 1; // switching call back based on suspension point
    }

    void _s2(){ // Operations b/w suspension point 1 & 2
        a = 10;
        int c = 10;

        sp = 2;
    }

    void _s3(){ // Operations after suspension point 2
        promise.return_value(a + b + c);  // After execution, coro_done = true;

        coro_done = true;
    }

    int a, b, c; // Local variables

    promise_type& promise;

    void (*s_cb)[] = {_s1, _s2, _s3}; // suspension point callbacks
    int sp{0}; // current suspension point(sp)
    bool coro_done{false};
};
```

- Now, when you call coroutine, you are basically creating a coroutine state object with a similar argument to that of coroutine.

```
auto mycoro = print_hello_world(5);

// transform into

print_hello_world    mycoro(5);
```

- As you might have guessed, due to rewriting coroutine into tiny functions, we don’t need a suspension mechanism anymore. And resumption can be implemented as:

```
void std::coroutine_handle::resume() {
    if(not print_hello_world::coro_done)
        print_hello_world::s_cb[sp]();
}
```

- So, when you call resume you are basically calling the suspension-point specific tiny function(technically its method) from a coroutine state object. That implies the creation of two stack frame i.e. `std::coroutine_handle::resume()` & `print_hello_world::_sX()`, where `X` stands for a suspension point number. Consequently not needing a separate stack & using the same stack as the caller. And this is why C++20 coroutines are stackless coroutine & faster.
- I agree that the above example is not compilable, don't cover edge cases & doesn't consider customized(i.e. awaiter object) coroutine behaviour. But, with the intuition we just built, you can easily create a mental model of coroutine & see the compiler perspective.
- And to test your intuition capability, just imagine **How you can transform the tiny functions for the coroutine that contains `co_await` expression in a for loop?**

Hypothesis 2

- Another usual way you can implement suspension-resumption is with the help of context switching APIs. For example, in the expansion of `co_wait <expr>` statement, I have mentioned the comment *"compiler added suspend/resume hook"* as below.

```
{
    auto&& awaiter = std::suspend_always{};
    if (!awaiter.await_ready()) {
        awaiter.await_suspend(std::coroutine_handle<> p);
        // compiler added suspend/resume hook
    }
    awaiter.await_resume();
}
```

- This is the place where your compiler adds its secret sauce to your code with context switching APIs. And rest is I think easy to imagine if you have read my earlier post Coroutine in C Language thoroughly.

Summary

Subroutine vs Coroutine

Subroutines	Coroutines	
call	calling convention i.e. foo()	calling convention i.e. foo()
suspend	Can't suspend	co_await/co_yield expression
resume	Can't suspend	coroutine_handle<>::resume()
return	return statement	co_return statement

Compiler Transformation Of Coroutine Related Keywords

```
co_return x;

// transforms into

__promise.return_value(x);
goto __final_suspend_label;
```

```
co_await y;

// transforms into

auto&& __awaitable = y;
if (__awaitable.await_ready()) {
    __awaitable.await_suspend();
    // compiler added suspend/resume hook
}
__awaitable.await_resume();
```

```
co_yield z;

// transforms into

co_await __promise.yield_value(z);
```

Parting Words With FAQs

- This is not just it, there are still many things that we haven't explored like exception handling, coroutine calling coroutine, etc.
- One unusual thing I observed about C++20 Coroutine is, You have to specify the return type even if you don't return anything from the coroutine. IMO, this is unusual & counter-intuitive.

What does it mean by stackfull & stackless coroutine?

- stackfull coroutines need a separate stack to be executed.
- stackless coroutine uses the same stack as of caller.

Difference between threads & coroutines?

- Coroutines are about your *programming model* and threads are about your *execution model*.
- co-routine can still do concurrency without scheduler overhead – it simply manages the context-switching itself.
- Another benefit is the much lower memory usage. With the threaded model, each thread needs to allocate its own stack, and so your memory usage grows linearly with the number of threads you have.

What does it mean by "coroutines are like lightweight threads"?

- First of all, coroutines & threads are a different entity. Because coroutine can be stackless & doesn't need OS intervention on schedule, it is lower on memory footprint & faster on execution as compared to thread. With C++, it's, even more, faster if you use Hypothesis 1.

How come C++ coroutine are stackless?

- This is already answered in Hypothesis 1. Still one more time.
- The compiler "returns" a handle to this dynamically allocated coroutine frame to the caller of the coroutine.
- When a caller calls `std::coroutine_handle::resume()` method, stackframe for `std::coroutine_handle::resume()` will be created which resumes the coroutine while processing all coroutine data on a dynamically allocated coroutine frame.

- And this is how coroutine is stackless in C++. Because it executes in the current stack only. Instead of a separate coroutine stack.