Introduction to C++ Coroutines

JAMES MCNELLIS
SENIOR SOFTWARE ENGINEER
MICROSOFT VISUAL C++

Motivation

WHY ADD COROUTINES AT ALL?

```cpp
int64_t tcp_reader(int64_t total) {
    std::array<char, 4096> buffer;

    tcp::connection the_connection = tcp::connect("127.0.0.1", 1337);
    for (;;) {
        int64_t bytes_read = the_connection.read(buffer.data(), buffer.size());
        total -= bytes_read;
        if (total <= 0 || bytes_read == 0) { return total; }
    }
}
```

```cpp
std::future<int64_t> tcp_reader(int64_t total) {
    struct reader_state {
        std::array<char, 4096> _buffer;
        int64_t                _total;
        tcp::connection        _connection;
        explicit reader_state(int64_t total) : _total(total) {}
    };

    auto state = std::make_shared<reader_state>(total);

    return tcp::connect("127.0.0.1", 1337).then(
        [state](std::future<tcp::connection> the_connection) {
            state->_connection = std::move(the_connection.get());
            return do_while([state]() -> std::future<bool> {
                if (state->_total <= 0) { return std::make_ready_future(false); }
                return state->conn.read(state->_buffer.data(), sizeof(state->_buffer)).then(
                    [state](std::future<int64_t> bytes_read_future) {
                        int64_t bytes_read = bytes_read_future.get();
                        if (bytes_read == 0) { return std::make_ready_future(false); }
                        state->_total -= bytes_read;
                        return std::make_ready_future(true);
                    });
            });
        });
}
```

```
std::future<int64_t> tcp_reader(int64_t total) {
    struct reader_state {
        std::array<char, 4096> _buffer;
        int64_t                _total;
        tcp::connection        _connection;
        explicit reader_state(int64_t total) : _total(total) {}
    };

    auto state = std::make_shared<reader_state>(total);

    return tcp::connect("127.0.0.1", 1337).then(
        [state](std::future<tcp::connection> the_connection) {
            state->_connection = std::move(the_connection.get());
            return do_while([state]() -> std::future<bool> {
                if (state->_total <= 0) { return std::make_ready_future(false); }
                return state->conn.read(state->_buffer.data(), sizeof(state->_buffer)).then(
                    [state](std::future<int64_t> bytes_read_future) {
                        int64_t bytes_read = bytes_read_future.get();
                        if (bytes_read == 0) { return std::make_ready_future(false); }
                        state->_total -= bytes_read;
                        return std::make_ready_
                    });
            });
        });
}
```

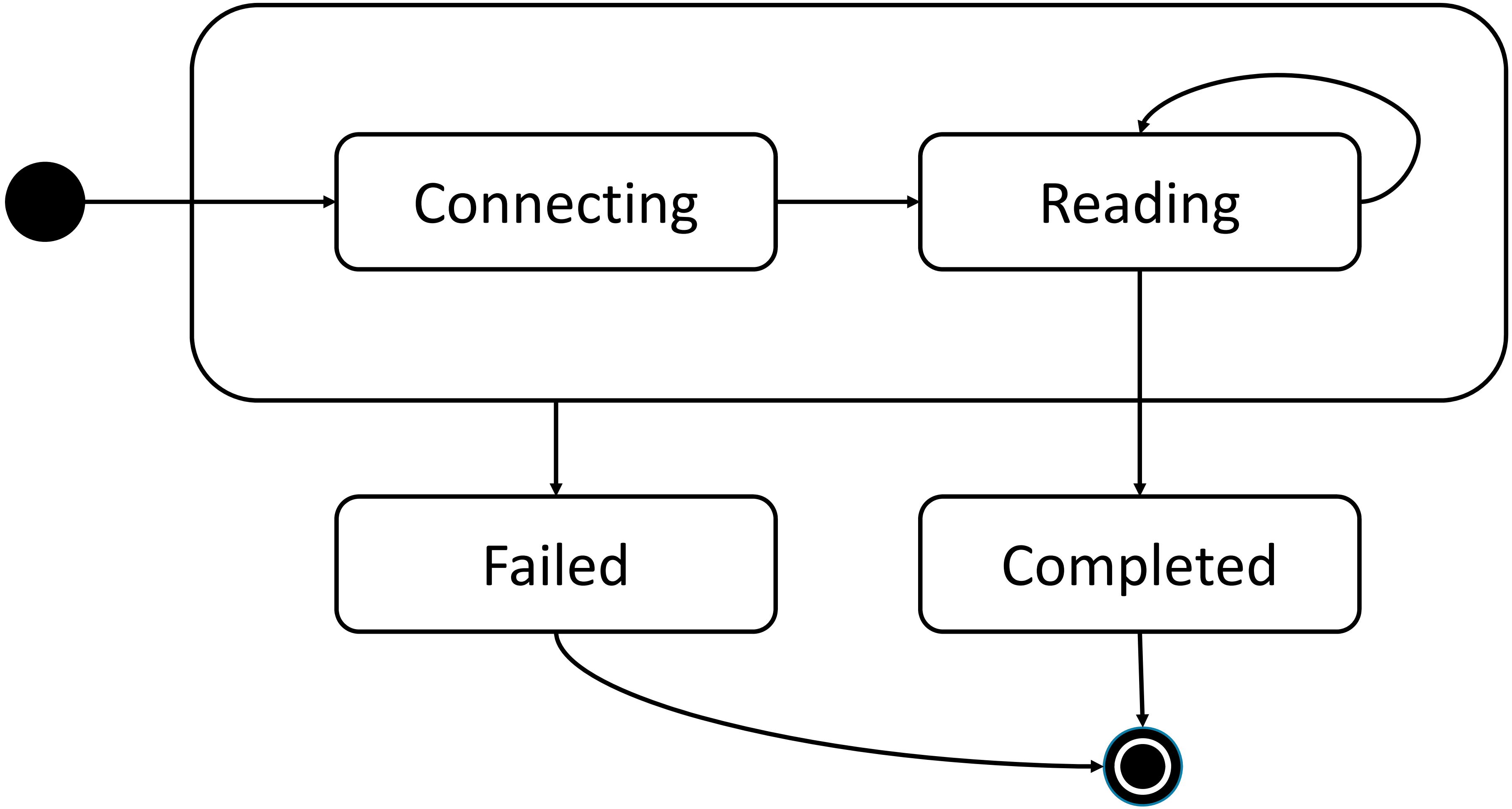
```
future<void> do_while(function<future<bool>()> body) {
    return body().then([=](future<bool> not_done) {
        return not_done.get() ? do_while(body) : make_ready_future();
    })
}
```

```cpp
int64_t tcp_reader(int64_t total) {
    std::array<char, 4096> buffer;

    tcp::connection the_connection = tcp::connect("127.0.0.1", 1337);
    for (;;) {
        int64_t bytes_read = the_connection.read(buffer.data(), buffer.size());
        total -= bytes_read;
        if (total <= 0 || bytes_read == 0) { return total; }
    }
}
```

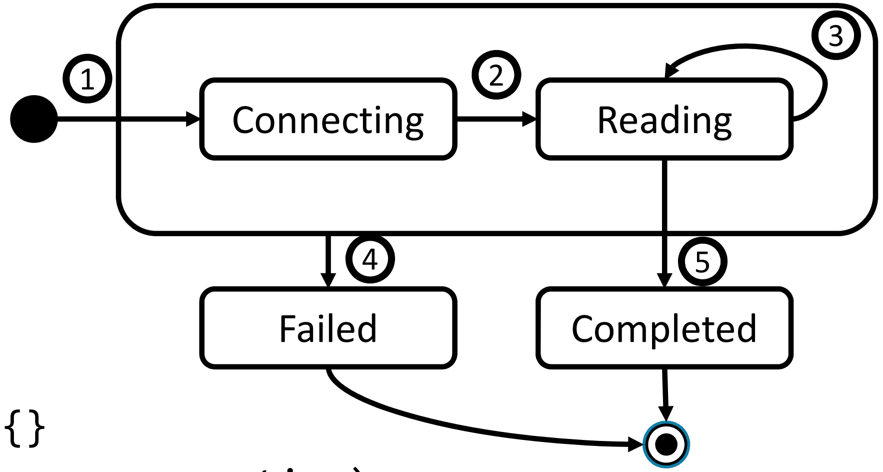
```cpp
std::future<int64_t> tcp_reader(int64_t total) {
    struct reader_state {
        std::array<char, 4096> _buffer;
        int64_t                _total;
        tcp::connection        _connection;
        explicit reader_state(int64_t total) : _total(total) {}
    };

    auto state = std::make_shared<reader_state>(total);

    return tcp::connect("127.0.0.1", 1337).then(
        [state](std::future<tcp::connection> the_connection) {
            state->_connection = std::move(the_connection.get());
            return do_while([state]() -> std::future<bool> {
                if (state->_total <= 0) { return std::make_ready_future(false); }
                return state->conn.read(state->_buffer.data(), sizeof(state->_buffer)).then(
                    [state](std::future<int64_t> bytes_read_future) {
                        int64_t bytes_read = bytes_read_future.get();
                        if (bytes_read == 0) { return std::make_ready_future(false); }
                        state->_total -= bytes_read;
                        return std::make_ready_future(true);
                    });
            });
        });
}
```

```
std::future<int64_t> tcp_reader(int64_t total) {
    struct reader_state {
        std::array<char, 4096> _buffer;
        int64_t                _total;
        tcp::connection        _connection;
        explicit reader_state(int64_t total) : _total(total) {}
    };

    auto state = std::make_shared<reader_state>(total);

    return tcp::connect("127.0.0.1", 1337).then(
        [state](std::future<tcp::connection> the_connection) {
            state->_connection = std::move(the_connection.get());
            return do_while([state]() -> std::future<bool> {
                if (state->_total <= 0) { return std::make_ready_future(false); }
                return state->conn.read(state->_buffer.data(), sizeof(state->_buffer)).then(
                    [state](std::future<int64_t> bytes_read_future) {
                        int64_t bytes_read = bytes_read_future.get();
                        if (bytes_read == 0) { return std::make_ready_future(false); }
                        state->_total -= bytes_read;
                        return std::make_ready_future(true);
                    });
            });
        }).then([state]{return std::make_ready_future(state->_total); });
}
```

Maybe a state machine will be simpler…


Connecting
Reading
Failed
Completed

```
class tcp_reader
{
    std::array<char, 4096> _buffer;
    tcp::connection        _connection;
    std::promise<int64_t>  _done;
    int64_t                _total;

    explicit tcp_reader(int64_t total) : _total(total) {}
  2 void on_connect(std::error_code ec, tcp::connection new_connection);
  3 void on_read(std::error_code ec, int64_t bytes_read);
  4 void on_error(std::error_code ec);
  5 void on_complete();

public:
  1 static std::future<int64_t> start(int64_t total);
};
```



```cpp
future<int64_t> tcp_reader::start(int64_t total) {
    auto p = std::make_unique<tcp_reader>(total);
    auto result = p->_done.get_future();
    tcp::connect("127.0.0.1", 1337, [raw = p.get()](auto ec, auto new_connection) {
        raw->on_connect(ec, std::move(new_connection));
    });
    p.release();
    return result;
}

void tcp_reader::on_connect(std::error_code ec, tcp::connection new_connection) {
    if (ec) { return on_error(ec); }
    _connection = std::move(new_connection);
    _connection.read(_buffer.data(), _buffer.size(),
        [this](std::error_code ec, int64_t bytes_read) {
            on_read(ec, bytes_read);
        });
}
```

```cpp
void tcp_reader::on_read(std::error_code ec, int64_t bytes_read) {
    if (ec) { return on_error(ec); }
    _total -= bytes_read;
    if (_total <= 0 || bytes_read == 0) { return on_complete(); }
    _connection.read(_buffer.data(), _buffer.size(),
        [this](std::error_code ec, int64_t bytes_read) {
            on_read(ec, bytes_read);
        });
}

void tcp_reader::on_error(std::error_code ec) {
    auto clean_me = std::unique_ptr<tcp_reader>(this);
    _done.set_exception(std::make_exception_ptr(std::system_error(ec)));
}

void tcp_reader::on_complete() {
    auto clean_me = std::unique_ptr<tcp_reader>(this);
    _done.set_value(_total);
}
```

What if…

```
auto tcp_reader(int64_t total) ->                int64_t
{
    std::array<char, 4096> buffer;

    tcp::connection the_connection =          tcp::connect("127.0.0.1", 1337);
    for (;;)
    {
        int64_t bytes_read =          the_connection.read(buffer.data(), buffer.size());
        total -= bytes_read;
        if (total <= 0 || bytes_read == 0) {    return total; }
    }
}
```

```cpp
auto tcp_reader(int64_t total) -> std::future<int64_t>
{
    std::array<char, 4096> buffer;

    tcp::connection the_connection = co_await tcp::connect("127.0.0.1", 1337);
    for (;;)
    {
        int64_t bytes_read = co_await the_connection.read(buffer.data(), buffer.size());
        total -= bytes_read;
        if (total <= 0 || bytes_read == 0) { co_return total; }
    }
}
```

```cpp
auto tcp_reader(int64_t total) -> std::future<int64_t>
{
    std::array<char, 4096> buffer;

    tcp::connection the_connection = co_await tcp::connect("127.0.0.1", 1337);
    for (;;)
    {
        int64_t bytes_read = co_await the_connection.read(buffer.data(), buffer.size());
        total -= bytes_read;
        if (total <= 0 || bytes_read == 0) { co_return total; }
    }
}
```

The Basics

What is a Coroutine?

A coroutine is a generalization of a subroutine

A subroutine…
- …can be invoked by its caller
- …can return control back to its caller

A coroutine has these properties, but also…
- …can suspend execution and return control to its caller
- …can resume execution after being suspended

In C++ (once this feature is added)…
- …both subroutines and coroutines are *functions*
- …a function can be either a subroutine *or* a coroutine

Subroutines and Coroutines

	Subroutine	Coroutine
Invoke	Function call, e.g. `f()`	Function call, e.g. `f()`
Return	`return` statement	`co_return` statement
Suspend		`co_await` expression
Resume		

(This table is incomplete; we'll be filling in a few more details as we go along…)

What makes a function a coroutine?

Is this function a coroutine?

```
std::future<int> compute_value();
```

Maybe. Maybe not.

Whether a function is a coroutine is an *implementation detail*.

- It’s not part of the type of a function
- It has no effect on the function *declaration* at all

What makes a function a coroutine?

A function is a coroutine if it contains…

- …a `co_return` statement,
- …a `co_await` expression,
- …a `co_yield` expression, or
- …a range-based for loop that uses `co_await`

Basically, a function is a coroutine if it uses any of the coroutine support features

What makes a function a coroutine?

```
std::future<int> compute_value()
{
    return std::async([]
    {
        return 30;
    });
}
```

```
std::future<int> compute_value()
{
    int result = co_await std::async([]
    {
        return 30;
    });

    co_return result;
}
```

What does co_await actually do?

```
auto result = co_await expression;
```

What does co_await actually do?

```
auto result = co_await expression;
```

```
auto&& __a = expression;
if (!__a.await_ready())
{
    __a.await_suspend(coroutine-handle);
    // ...suspend/resume point...
}

auto result = __a.await_resume();
```

What does co_await actually do?

```
struct awaitable_concept
{
    bool await_ready();
    void await_suspend(coroutine_handle<>);
    T    await_resume();
};
```

The simplest awaitable: suspend_always

```
struct suspend_always
{
    bool await_ready() noexcept
    {
        return false;
    }

    void await_suspend(coroutine_handle<>) noexcept { }
    void await_resume() noexcept { }
};
```

The simplest awaitable: suspend_always

```
return_type my_coroutine()
{
    cout << "my_coroutine about to suspend\n";

    co_await suspend_always{}; // This will suspend the coroutine and return
                               // control back to its caller

    cout << "my_coroutine was resumed\n";
}
```

Another simple awaitable: suspend_never

```
struct suspend_never
{
    bool await_ready() noexcept
    {
        return true;
    }

    void await_suspend(coroutine_handle<>) noexcept { }
    void await_resume() noexcept { }
};
```

Another simple awaitable: suspend_never

```
return_type my_coroutine()
{
    cout << "my_coroutine before 'no-op' await\n";

    co_await suspend_never{}; // This will not suspend the coroutine and will
                              // allow the coroutine to continue execution.

    cout << "my_coroutine after 'no-op' await\n";
}
```

So that's the first half…

When a coroutine is executing, it uses co_await to suspend itself and return control to its caller

How does its caller resume a coroutine?

What happens when you invoke a function?

When you call a function, the compiler has to “construct” a *stack frame*

The stack frame includes space for…

- …arguments
- …local variables
- …the return value
- …storage for volatile registers (maybe)

What happens when you invoke a coroutine?

The compiler needs to construct a *coroutine frame* that contains space for…

- …the formal parameters
- …all local variables
- …selected temporaries
- …execution state for when the coroutine is suspended (registers, instruction pointer, etc.)
- …the “promise” that is used to return a value or values

In general, the coroutine frame must be dynamically allocated

- the coroutine loses use of the stack when it is suspended
- operator new is used, but it can be overloaded for specific coroutines, to allow allocation customization.

Creation of the coroutine frame occurs before the coroutine starts running

- just like creation of a stack frame for an ordinary function

The compiler “returns” a handle to this coroutine frame to the caller of the coroutine

```cpp
template <>
struct coroutine_handle<void>
{
    // ...
};

template <typename Promise>
struct coroutine_handle
    : coroutine_handle<void>
{
    // ...
};
```

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

    static coroutine_handle from_address(void* _Addr) noexcept;
    void* to_address() const noexcept;

    void operator()() const;
    void resume() const;

    void destroy();

    bool done() const;
};
```

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

    static coroutine_handle from_promise(Promise&) noexcept;
};
```

Let’s Build
A Simple Coroutine

```
resumable_thing counter()
{
    cout << "counter: called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter: resumed (#" << i << ")\n";
    }
}

int main()
{
    cout << "main:    calling counter\n";
    resumable_thing the_counter = counter();
    cout << "main:    resuming counter\n";
    the_counter.resume();
    the_counter.resume();
    cout << "main:    done\n";
}
```

```cpp
resumable_thing counter()
{
    cout << "counter: called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter: resumed (#" << i << ")\n";
    }
}

int main()
{
    cout << "main:    calling counter\n";
    resumable_thing the_counter = counter();
    cout << "main:    resuming counter\n";
    the_counter.resume();
    the_counter.resume();
    cout << "main:    done\n";
}
```

```
main:    calling counter
counter: called
main:    resuming counter
counter: resumed (#1)
counter: resumed (#2)
main:    done
```

```cpp
struct resumable_thing
{
    struct promise_type;

    coroutine_handle<promise_type> _coroutine = nullptr;

    explicit resumable_thing(coroutine_handle<promise_type> coroutine)
        : _coroutine(coroutine) { }

    ~resumable_thing()
    {
        if (_coroutine) { _coroutine.destroy(); }
    }

    void resume() const { _coroutine.resume(); }

    // ...
};
```

```cpp
struct resumable_thing
{
    // ...
    resumable_thing() = default;
    resumable_thing(resumable_thing const&) = delete;
    resumable_thing& operator=(resumable_thing const&) = delete;

    resumable_thing(resumable_thing&& other)
        : _coroutine(other._coroutine) {
        other._coroutine = nullptr;
    }

    resumable_thing& operator=(resumable_thing&& other) {
        if (&other != this) {
            _coroutine = other._coroutine;
            other._coroutine = nullptr;
        }
    }
};
```

```cpp
struct resumable_thing
{
    struct promise_type
    {
        resumable_thing get_return_object()
        {
            return resumable_thing(coroutine_handle<promise_type>::from_promise(this));
        }

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

        void return_void() { }
    };

    // ...
};
```

```cpp
resumable_thing counter()
{
    cout << "counter: called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter: resumed (#" << i << ")\n";
    }
}
```

```cpp
resumable_thing counter()
{
    cout << "counter: called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter: resumed (#" << i << ")\n";
    }
}

struct __counter_context
{
    resumable_thing::promise_type _promise;
    unsigned                      _i;
    void*                         _instruction_pointer;
    // storage for registers, etc.
};
```

```
resumable_thing counter()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "counter: called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter: resumed (#" << i << ")\n";
    }

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

```cpp
resumable_thing named_counter(std::string name)
{
    cout << "counter(" << name << ") was called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter(" << name << ") resumed #" << i << '\n';
    }
}

int main()
{
    resumable_thing counter_a = named_counter("a");
    resumable_thing counter_b = named_counter("b");
    counter_a.resume();
    counter_b.resume();
    counter_b.resume();
    counter_a.resume();
}
```

```cpp
resumable_thing named_counter(std::string name)
{
    cout << "counter(" << name << ") was called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter(" << name << ") resumed #" << i << '\n';
    }
}

int main()
{
    resumable_thing counter_a = named_counter("a");
    resumable_thing counter_b = named_counter("b");
    counter_a.resume();
    counter_b.resume();
    counter_b.resume();
    counter_a.resume();
}
```

```
counter(a) was called
counter(b) was called
counter(a) resumed #1
counter(b) resumed #1
counter(b) resumed #2
counter(a) resumed #2
```

Subroutines and Coroutines

	Subroutine	Coroutine
Invoke	Function call, e.g. `f()`	Function call, e.g. `f()`
Return	`return` statement	`co_return` statement
Suspend		`co_await` expression
Resume		`coroutine_handle<>::resume()`

(This table is incomplete; we'll be filling in a few more details as we go along…)

Returning from a Coroutine

Returning from a Coroutine

```
std::future<int> compute_value()
{
    int result = co_await std::async([]
    {
        return 30;
    });

    co_return result;
}
```

Returning from a Coroutine

```
std::future<int> compute_value()
{
    int result = co_await std::async([]
    {
        return 30;
    });

    co_return result;
}
```

What’s in a Promise?

```
struct promise_type
{
    resumable_thing get_return_object();

    auto initial_suspend();
    auto final_suspend();

    void return_void();          // called for a co_return with no argument
                                 // (or falling off the end of a coroutine)

    void return_value(T value) // called for a co_return with argument
};
```

```
resumable_thing get_value()
{
    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";
    co_return 30;
}

int main()
{
    cout << "main:      calling get_value\n";
    resumable_thing value = get_value();
    cout << "main:      resuming get_value\n";
    value.resume();
    cout << "main:      value was " << value.get() << '\n';
}
```

```
resumable_thing get_value()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";

    co_return 30;


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

```
resumable_thing get_value()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";

    co_return 30;


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

```
resumable_thing get_value()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";

    __context->_promise.return_value(30);
    goto __final_suspend_label;

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

```
struct resumable_thing
{
    struct promise_type
    {
        int _value;

        resumable_thing get_return_object()
        {
            return resumable_thing(coroutine_handle<promise_type>::from_promise(this));
        }

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

        void return_value(int value) { _value = value; }
    };

    int get() { return _coroutine.promise()._value; }
};
```

```cpp
resumable_thing get_value()
{
    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";
    co_return 30;
}

int main()
{
    cout << "main:      calling get_value\n";
    resumable_thing value = get_value();
    cout << "main:      resuming get_value\n";
    value.resume();
    cout << "main:      value was " << value.get() << '\n';
}
```

```
main:      calling get_value
get_value: called
main:      resuming get_value
get_value: resumed
main:      value was 7059560
```

Coroutine Lifetime

A coroutine comes into existence when it is called

- This is when the compiler creates the *coroutine context*

A coroutine is destroyed when…

- …the final-suspend is resumed, or
- …`coroutine_handle<>::destroy` is called,

…whichever happens *first*.

```
resumable_thing get_value()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";

    __context->_promise.return_value(30);
    goto __final_suspend_label;

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

```
auto final_suspend()   { return suspend_never{};  }
```

```
struct resumable_thing
{
    struct promise_type
    {
        int _value;

        resumable_thing get_return_object()
        {
            return resumable_thing(coroutine_handle<promise_type>::from_promise(this));
        }

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

        void return_value(int value) { _value = value; }
    };

    int get() { return _coroutine.promise()._value; }
};
```

```cpp
struct resumable_thing
{
    struct promise_type
    {
        int _value;

        resumable_thing get_return_object()
        {
            return resumable_thing(coroutine_handle<promise_type>::from_promise(this));
        }

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

        void return_value(int value) { _value = value; }
    };

    int get() { return _coroutine.promise()._value; }
};
```

```
resumable_thing get_value()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";

    __context->_promise.return_value(30);
    goto __final_suspend_label;

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

```
auto final_suspend()   { return suspend_always{}; }
```

```cpp
resumable_thing get_value()
{
    cout << "get_value: called\n";
    co_await suspend_always{};
    cout << "get_value: resumed\n";
    co_return 30;
}

int main()
{
    cout << "main:      calling get_value\n";
    resumable_thing value = get_value();
    cout << "main:      resuming get_value\n";
    value.resume();
    cout << "main:      value was " << value.get_value() << '\n';
}
```

```
main:      calling get_value
get_value: called
main:      resuming get_value
get_value: resumed
main:      value was 30
```

```cpp
~resumable_thing()
{
    if (_coroutine) { _coroutine.destroy(); }
}
```

```cpp
resumable_thing get_value()
{
    print_when_destroyed a("get_value: a destroyed");
    co_await suspend_always{};
    cout << "get_value: resumed\n";
    print_when_destroyed b("get_value: b destroyed");
    co_return 30;
}

int main()
{
    cout << "main:      calling get_value\n";
    resumable_thing value = get_value();
    cout << "main:      resuming get_value\n";
    value.resume();
    cout << "main:      value was " << value.get() << '\n';
}
```

```
main:      calling get_value
main:      resuming get_value
get_value: resumed
get_value: b destroyed
get_value: a destroyed
main:      value was 30
```

```
resumable_thing get_value()
{
    print_when_destroyed a("get_value: a destroyed");
    co_await suspend_always{};
    cout << "get_value: resumed\n";
    print_when_destroyed b("get_value: b destroyed");
    co_return 30;
}

int main()
{
    cout << "main:       calling get_value\n";
    resumable_thing value = get_value();
    cout << "main:       resuming get_value\n";
    value.resume();
    cout << "main:       value was " << value.get() << '\n';
}
```

```
main:       calling get_value
main:       value was 0
get_value: a destroyed
```

Coroutine Lifetime

A coroutine is destroyed when…

- …the final-suspend is resumed, or
- …`coroutine_handle<>::destroy` is called,

…whichever happens *first*.

When a coroutine is destroyed, it cleans up local variables

- …but only those that were initialized prior to the last suspension point

Subroutines and Coroutines

	Subroutine	Coroutine
Invoke	Function call, e.g. `f()`	Function call, e.g. `f()`
Return	`return` statement	`co_return` statement
Suspend		`co_await` expression
Resume		`coroutine_handle<>::resume()`

(This table is incomplete; we'll be filling in a few more details as we go along…)

How about something that's actually useful…

Let’s look at future…

```
future<int> compute_value()
{
    int result = co_await async([]
    {
        return 30;
    });

    co_return result;
}
```

Let’s look at future…

```
future<int> compute_value()
{
    int result = co_await async([]
    {
        return 30;
    });

    co_return result;
}
```

Let’s look at future…

```
template <typename T>
cla      ure
{
    //

    stru          e_type
    {


};
```

coroutine_traits<T>

```
template <typename Return, typename... Arguments>
struct coroutine_traits;
```

coroutine_traits<T>

```
template <typename Return, typename... Arguments>
struct coroutine_traits
{
    using promise_type = typename Return::promise_type;
};
```

```
template <typename T, typename... Arguments>
struct coroutine_traits<future<T>, Arguments...>
{
    struct promise_type
    {
        promise<T> _promise;

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

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

        template <typename U>
        void return_value(U&& value) { _promise.set_value(std::forward<U>(value)); }

        void set_exception(std::exception_ptr ex) { _promise.set_exception(std::move(ex)); }
    };
};
```

Let’s look at future…

```
future<int> compute_value()
{
    int result = co_await async([]
    {
        return 30;
    });

    co_return result;
}
```

Let’s look at future…

```
template <typename T>
cl    future
{
    //

    bo        ready();
    v    awa     spend(coroutine_handle<>);
       d await     me();
};
```

```cpp
template <typename T>
bool await_ready(future<T>& f) { return f.is_ready()); }

template <typename T>
void await_suspend(future<T>& f, coroutine_handle<> ch)
{
    f.then([ch](future<T>& f)
    {
        ch.resume();
    });
}

template <typename T>
auto await_resume(future<T>& f) { return f.get()); }
```

```cpp
template <typename T>
bool await_ready(future<T>& f) { return f.is_ready()); }

template <typename T>
void await_suspend(future<T>& f, coroutine_handle<> ch)
{
    f.then([ch](future<T>& f)
    {
        ch.resume();
    });
}

template <typename T>
auto await_resume(future<T>& f) { return f.get()); }
```

Let’s look at future…

```
future<int> compute_value()
{
    int result = co_await async([]
    {
        return 30;
    });

    co_return result;
}
```

Yielding

```cpp
generator<int> integers(int first, int last)
{
    for (int i = first; i <= last; ++i)
    {
        co_yield i;
    }
}

int main()
{

    for (int x : integers(1, 5))
    {
        cout << x << '\n';
    }
}
```

```
1
2
3
4
5
```

```cpp
generator<int> integers(int first, int last)
{
    for (int i = first; i <= last; ++i)
    {
        co_yield i;
    }
}

int main()
{
    generator<int> the_integers = integers(1, 5);
    for (auto it = the_integers.begin(); it != the_integers.end(); ++it)
    {
        cout << *it << '\n';
    }
}
```

```
1
2
3
4
5
```

```cpp
generator<int> integers(int first, int last)
{
    for (int i = first; i <= last; ++i)
    {
        co_yield i;
    }
}
```

```cpp
generator<int> integers(int first, int last)
{
    for (int i = first; i <= last; ++i)
    {
        co_await __promise.yield_value(i);
    }
}
```

```
struct int_generator {
    struct promise_type {
        int const* _current;

        int_generator get_return_object() {
            return int_generator(coroutine_handle<promise_type>::from_promise(this));
        }

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

        auto yield_value(int const& value) {
            _current = &value;
            return suspend_always{};
        }
    };
};
```

```
struct int_generator
{
    struct iterator;

    iterator begin()
    {
        if (_coroutine)
        {
            _coroutine.resume();
            if (_coroutine.done()) { return end(); }
        }

        return iterator(_coroutine);
    }

    iterator end() { return iterator{}; }
    // ...
};
```

```cpp
struct int_generator
{
    struct iterator : std::iterator<input_iterator_tag, int>
    {
        coroutine_handle<promise_type> _coroutine;

        iterator& operator++()
        {
            _coroutine.resume();
            if (_coroutine.done()) { _coroutine = nullptr; }
            return *this;
        }

        int const& operator*() const
        {
            return *_coroutine.promise()._current;
        }
    };
};
```

```cpp
int_generator integers(int first, int last)
{
    for (int i = first; i <= last; ++i)
    {
        co_yield i;
    }
}

int main()
{

    for (int x : integers(1, 5))
    {
        cout << x << '\n';
    }
}
```

```
1
2
3
4
5
```

Summary

Subroutines and Coroutines

	Subroutine	Coroutine
Invoke	Function call, e.g. `f()`	Function call, e.g. `f()`
Return	`return` statement	`co_return` statement
Suspend		`co_await` expression `co_yield` expression
Resume		`coroutine_handle<>::resume()`

Coroutine Control Flow

Statement/Expression...	Equivalent to...
`co_return x;`	`__promise.return_value(x);` `goto __final_suspend_label;`

`co_await y`	`auto&& __awaitable = y;`
	`if (__awaitable.await_ready())`
	`{`
	`    __awaitable.await_suspend();`
	`    // ...suspend/resume point...`
	`}`
	
	`__awaitable.await_resume();`

`co_yield z`	`co_await __promise.yield_value(z)`

```
resumable_thing counter()
{
    __counter_context* __context = new __counter_context{};
    __return = __context->_promise.get_return_object();
    co_await   __context->_promise.initial_suspend();

    cout << "counter: called\n";
    for (unsigned i = 1; ; ++i)
    {
        co_await suspend_always{};
        cout << "counter: resumed\n";
    }

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

Design Principles

Scalable, to **b**illions of concurrent coroutines

Efficient: Suspend/resume operations comparable in cost to function call overhead

Open-Ended: Library designers can develop coroutine libraries exposing various high-level semantics, including generators, goroutines, tasks, and more

Seamless Interaction with existing facilities with no overhead.

Usable in environments where exceptions are forbidden or not available

References

WG21 Papers:

- N4402: Resumable Functions (revision 4) (Gor Nishanov, Jim Radigan)
- P0057R1: Wording for Coroutines (Gor Nishanov, Jens Maurer, Richard Smith)

Other Talks (All Available on YouTube):

- Introduction to C++ Coroutines (James McNellis, Meeting C++ 2015)
- C++ Coroutines: A Negative Overhead Abstraction (Gor Nishanov, CppCon 2015)
- Await 2.0: Stackless Resumable Functions (Gor Nishanov, CppCon 2014)

\0