
Core C++ 2025
19 Oct. 2025 :: Tel-Aviv
COROUTINES 101
Omer Anson

OUTLINE

- Why
- What
- How
- Examples

WHY

- Why use coroutines?
  - Provide parallel computaion feel
    - Allows scheduling in and out mid-processing
  - Lightweight
  - Single thread
    - No need for pesky synchronisations
  - Single process
    - Same memory space

SIMPLE PARSER EXAMPLE

- Provide parallel computaion feel - What does this mean?
- Let’s look at this parser example

- Suppose a parser API:

```
Tree parse(std::function<std::string_view()> more_data);
    // Repeatedly calls more_data for the input
    // Generate a tree
    // And return it
```

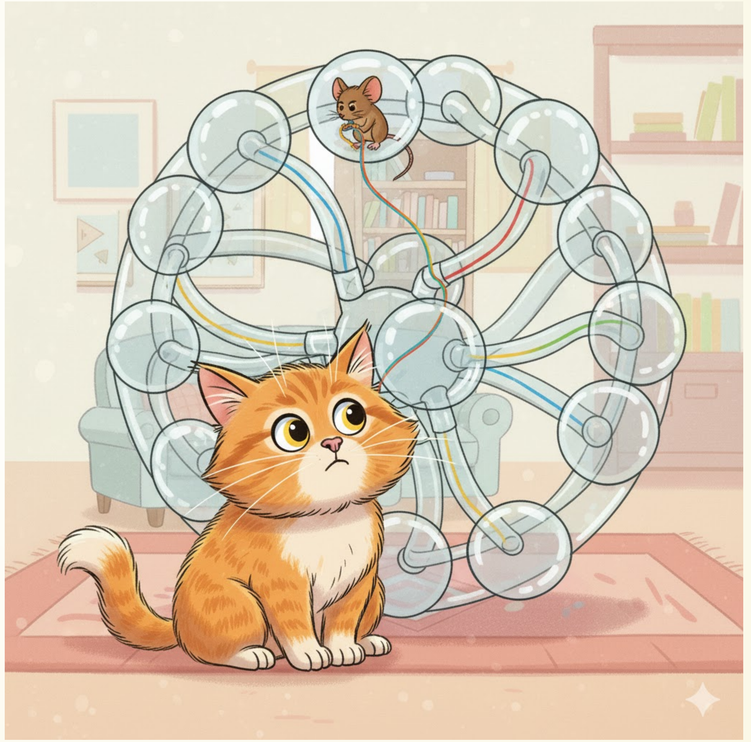
- But we want this API:

```
class Parser {
public:
    void push(std::string_view value);
    Tree value();
private:
  // ...
};
```

- Maybe because we only get our data a bit at a time.

STATE MACHINES AND PARSERS

- Often maintain a stack
- Sometimes on the call stack
- But they let you configure the input
- You can’t stop them
  - but they call you for more data



ACTOR MODEL

- Everything is an actor
- Actors schedule work for each other
- Actors create new actors
- No sequence assumptions

GENERATORS

- Similar to what views give today
- Generate an infinite list
- Iterate and operator on an infinite list

WHAT

So what are they really?

- Co-operative multitasking
- Single process
- Share memory
- Share CPU

WHAT - NOT THREADS

- Coroutines are not threads
- They do not run in parallel
- They are not pre-emptive

What does this mean?

- Sharing resources is easier
- No locking
- *you* decide when you yield

WHAT - NOT PROCESSES

- In general, the kernel is not part of the loop
- The kernel does not know you're hopping stacks

What does this mean?

- Kernel resources are shared
- Less information from outside
  - Coroutines do not appear in /proc

WHAT - NOT SUBROUTINES

- You decide when to yield
- You don’t have to run from start to finish
  - You can take little breaks

What does this mean?

- A coroutine can stop in the middle to ask for more info
- Or provide partial results
  - Maybe an element in an infinite list
  - Maybe an intermediate value in a converging calculation

HOW

We'll explore two APIs:

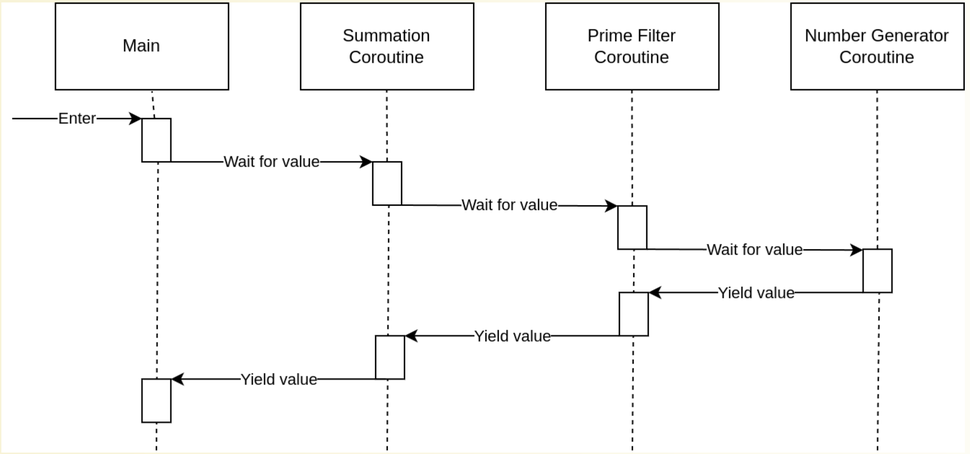
- Boost::coroutines2
- std::coroutines

By example:

- Sum all prime numbers. Once we pass 100, exit.
- Print everything along the way

Obviously, we won't go into the full detail. Read the relevant docs for more info

HOW - DESIGN



- Activation shows where the CPU is
- No two activations run in parallel
- await and yield suspend one coroutine and resume another

HOW - BOOST::COROUTINES2 - API

(Illustrative) API to create your coroutine:

```cpp
// Everything is defined here
#include <boost/coroutine2/all.hpp>

// Create a pull_type coroutine
boost::coroutines2::coroutine<T>::pull_type (
    std::function<void(boost::coroutines2::coroutine<T>::push_type &)> fn);
// Create a push_type coroutine
boost::coroutines2::coroutine<T>::push_type (
    std::function<void(boost::coroutines2::coroutine<T>::pull_type &)> fn);
```

- Creates a `pull_type`/`push_type` pair.
  - One for the caller, the other for the coroutine
- Calling one yields to the other.
- The coroutine continues executing from current location
  - The main context also counts as a coroutine
- Call a `push_type` with a value.
  - This “yields” the value.
- Call a `pull_type` (and then call `get()`)
  - This receives the value.

HOW - BOOST::COROUTINES2 - USAGE

Let’s create our number generator:

```
void generate_numbers(boost::coroutines2::coroutine<unsigned>::push_type& yield) {
    unsigned value = 0;
    while (true) {
        ++value;
        std::cout << "genenrate_numbers: --> " << value << std::endl;
        yield(value);
    }
}
```

HOW - BOOST::COROUTINES2 - USAGE

Now our prime tester:

```
bool is_prime(unsigned);  // I'm not implementing this
struct PrimeTesterCoro {
    boost::coroutines2::coroutine<unsigned>::pull_type &m_source;
    ~PrimeTesterCoro() { std::cout << "PrimeTesterCoro::~PrimeTesterCoro" << std::endl; }
    void operator()(boost::coroutines2::coroutine<unsigned>::push_type& yield) {
        for (unsigned i : m_source) {
            std::cout << "PrimeTesterCoro(): <-- " << i << std::endl;
            if (is_prime(i)) {
                std::cout << "PrimeTesterCoro(): --> " << i << std::endl;
                yield(i);
            }
        }
    }
};
```

HOW - BOOST::COROUTINES2 - USAGE

And our main loop:

```
int main() {
    boost::coroutines2::coroutine<unsigned>::pull_type generator{generate_numbers};
    boost::coroutines2::coroutine<unsigned>::pull_type prime_tester{PrimeTesterCoro{generator}};
    boost::coroutines2::coroutine<unsigned>::pull_type summer{
        [&prime_tester](boost::coroutines2::coroutine<unsigned>::push_type &yield) {
            BOOST_SCOPE_EXIT(void) { std::cout << "Clean shutdown" << std::endl; } BOOST_SCOPE_EXIT_END
            unsigned sum = 0;
            for (unsigned i : prime_tester) {
                std::cout << "summation lambda:  <-- " << i << std::endl;
                sum += i;
                std::cout << "summation lambda:  --> " << sum << std::endl;
                yield(sum);
            }
        }
    };
    while (true) {
        if (not summer()) { /* Handle error. */ }
        unsigned sum{summer.get()};
        std::cout << "main:              <-- " << sum << std::endl;
        if (sum > 100) { break; }
    }
    return 0;
}
```

HOW - BOOST::COROUTINES2 - OUTPUT

```
genenrate_numbers: --> 1
PrimeTesterCoro::~PrimeTesterCoro
PrimeTesterCoro(): <-- 1
genenrate_numbers: --> 2
PrimeTesterCoro(): <-- 2
PrimeTesterCoro(): --> 2
PrimeTesterCoro::~PrimeTesterCoro
summation lambda:  <-- 2
summation lambda:  --> 2
main:              <-- 2
genenrate_numbers: --> 3
PrimeTesterCoro(): <-- 3
PrimeTesterCoro(): --> 3
summation lambda:  <-- 3
summation lambda:  --> 5
main:              <-- 5
...
genenrate_numbers: --> 22
PrimeTesterCoro(): <-- 22
genenrate_numbers: --> 23
PrimeTesterCoro(): <-- 23
PrimeTesterCoro(): --> 23
summation lambda:  <-- 23
summation lambda:  --> 100
main:              <-- 100
main: Exit
Clean shutdown
PrimeTesterCoro::~PrimeTesterCoro
PrimeTesterCoro::~PrimeTesterCoro
```

HOW - STD::COROUTINES - USAGE

- Available since C++20
- Requires defining a coroutine and promise structure:

```cpp
template <typename T> struct coroutine {
    struct promise_type {
        coroutine<T> get_return_object() { return {std::coroutine_handle<promise_type>::from_promise(*this)}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
        template <typename U>
        std::suspend_always yield_value(U && value) { m_value = std::forward<U>(value); return {}; }
        void return_void() {}
        T m_value{};
    };
    coroutine(std::coroutine_handle<promise_type> handle) : m_handle(handle) {}
    coroutine(const coroutine &) = delete;
    coroutine(coroutine &&) = delete;
    ~coroutine() { m_handle.destroy(); }
    std::coroutine_handle<promise_type> m_handle;
    // Resume the coroutine, and return the yielded value on next suspend
    const T& operator()() {
        m_handle();
        return m_handle.promise().m_value;
    }
};
```

HOW - STD::COROUTINES - USAGE

- That was the hard part. Now our number generator:

```
coroutine<unsigned> generate_numbers() {
    unsigned t{};
    while (true) {
        ++t;
        std::cout << "generate_numbers --> " << t << std::endl;
        co_yield t;
    }
}
```

HOW - STD::COROUTINES - USAGE

- Our prime tester:

```
bool is_prime(unsigned);  // I'm *still* not implementing this
struct PrimeTesterCoro {
    coroutine<unsigned> &m_source;
    ~PrimeTesterCoro() { std::cout << "PrimeTesterCoro::~PrimeTesterCoro" << std::endl; }
    coroutine<unsigned> operator()() {
        while (true) {
            unsigned i = m_source();
            std::cout << "PrimeTesterCoro: <-- " << i << std::endl;
            if (is_prime(i)) {
                std::cout << "PrimeTesterCoro: --> " << i << std::endl;
                co_yield i;
            }
        }
    }
};
```

HOW - STD::COROUTINES - USAGE

- And our main:

```
int main() {
    coroutine<unsigned> generator = generate_numbers<unsigned>();
    PrimeTesterCoro prime_tester{generator};
    // Must keep prime_tester, otherwise the coroutine is dangling
    coroutine<unsigned> prime_tester_coro = prime_tester();
    coroutine<unsigned> summer = [&prime_tester_coro]() -> coroutine<unsigned> {
        BOOST_SCOPE_EXIT(void) { std::cout << "Clean shutdown" << std::endl; } BOOST_SCOPE_EXIT_END
        unsigned sum{};
        while (true) {
            unsigned i = prime_tester_coro();
            std::cout << "summation lambda:  <-- " << i << std::endl;
            sum += i;
            std::cout << "summation lambda:  --> " << sum << std::endl;
            co_yield sum;
        }
    }();
    while (true) {
        unsigned sum{summer()};
        std::cout << "main:              <-- " << sum << std::endl;
        if (sum > 100) { break; }
    }
    return 0;
}
```

WHEN DO WE NEED A STACK?

- Boost's coroutines2 supports stack.
- Let's return to our parser function:

```
Tree parse(std::function<std::string_view()> more_data);
```

- Note how the parser has no idea coroutines are involved
- And we can't change it
  - It isn't ours

WHEN DO WE NEED A STACK?

- And we want to push the values from main
- We use coroutines:

```
struct Praser {
    Tree m_tree;
    boost::coroutines2::coroutine<std::string_type>::push_type m_parser{
        [&](boost::coroutines2::coroutine<std::string_view>::pull_type &source) {
            m_tree = parse([&source]() -> unsigned { source(); return source.get(); } );
        }};
    void push(std::string_view value) { m_parser(value); }
    Tree value() {
        m_parser(""); // Mark end of stream or whatever
        return m_tree;
    }
};
```

- Note that only when the parser returns, `m_tree` is updated.
- Here we use a member to return the value
  - We don’t need to worry about locking
    - Because coroutines are not threads (Yay!)

WHEN DO WE NEED A STACK?

- And then we can use it the way we want

```
int main() {
    Parser parser;
    while (is.good()) {
        std::string str(1024, '\0');
        is.read(&srd[0], 1024);
        parser.add(str);
    }
    Tree tree = parser.value();
    // ...
}
```

- As we've seen before:
  - When `main` pushes a value, the parser coroutine is resumed
  - Calling `value()` notifies the coroutine to complete
    - And sets the return value
    - And then returns it

WARNINGS!

- Dangling references
- Non-atomic operations

EXAMPLES

- Generators (We've done this)
- Parsers (We've done this too!)
- Actor model
- Mathematical Software

EXAMPLES - ACTORS

- Everything is an actor

```
struct Actor {
    coro_type::push_type & do_task; // The coroutine has access to the scheduler
    bool can_handle(const Task & task) const;
    virtual void operator()(Task & task) { do_task(task); }
};
```

- And scheduler:

```
struct Scheduler {
    void schedule(Task task) { m_tasks.emplace_back(std::move(task)); }
    void run();
    std::list<coro_type::push_type> m_actors;
    std::list<Task> m_tasks;
};
```

EXAMPLES - ACTORS

- Where the `run` function is nice and simple:

```
void Scheduler::run() {
    while (not m_tasks.empty()) {
        Task task = std::move(m_tasks.front());
        m_tasks.pop_front();
        for (auto & actor : m_actors) {
            if (actor.can_handle(task)) {
                actor(task);
                break;
            }
        }
    }
}
```

- Why does this works?
  - The call to `actor(task)` suspends the coroutine with run
    - The coroutine contains its own state
    - The scheduler doesn't need to know about it
  - When it resumes, it takes the next task of the queue
  - We don't need synchronisation - single thread
  - We don't need complex message passing - single process

EXAMPLE - MATHEMATICAL SOFTWARE

- Coroutines can be used to add control points for mathematical software.
- Consider the following algorithm for finding the square root:

```
double square_root(double x, double epsilon) {
    double result = x;
    while (abs(result*result - x) > epsilon) {
        result = (result + (x/result))/2;
    }
    return result;
}
```

- With coroutines, this can be written as:

```
struct square_root {
    double x;
    void operator()(boost::coroutines2::coroutine<double>::push_type & yield) {
        double result = x;
        do {
            result = (result + (x/result))/2;
        } while (yield(result));
    }
}
```

- The caller can verify the process converges
- The caller can decide on-the-fly when the result is good enough

THANK YOU