ACCU 2022

COROUTINES: C++ VS RUST

JONATHAN MÜLLER

Motivation: Guard Patrol

Guard AI in an RPG: Patrol between two points.

#

Motivation: Guard Patrol

Guard AI in an RPG: Patrol between two points.

#

Motivation: Guard Patrol

Guard AI in an RPG: Patrol between two points.

#

Motivation: Guard Patrol

Guard AI in an RPG: Patrol between two points.

#

Motivation: Guard Patrol

Guard AI in an RPG: Patrol between two points.

#

Guard Patrol: Naive loop

vz35KnbEh

```
void patrol(int& pos, int start, int end)
{
    pos = start;
    auto dir = +1;
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        pos += dir;
        if (pos == end)
            dir = -1;
        if (pos == start)
            dir = 1;
    }
}
```

Guard Patrol: Naive loop

vz35KnbEh

```
std::atomic<int> pos(0);
std::thread thr(patrol, std::ref(pos), 0, 10);

while (true)
{
    auto cur_pos = pos.load();
    render(cur_pos);
}
```

Guard Patrol: State machine

dsrbnb46z

```
struct state
{
    int* pos;
    int dir;
    int start, end;
};
```

```
state patrol_init(int& pos, int start, int end)
{
    pos = start;
    return {&pos, +1, start, end};
}
```

Guard Patrol: State machine

dsrbnb46z

```
struct state
{
    int* pos;
    int dir;
    int start, end;
};
```

```
void patrol_update(state& state)
{
    *state.pos += state.dir;
    if (*state.pos == state.end)
        state.dir = -1;
    if (*state.pos == state.start)
        state.dir = +1;
}
```

Guard Patrol: State machine

dsrbnb46z

```
auto pos = 0;
auto state = patrol_init(pos, 0, 10);
while (true)
{
    auto cur_pos = pos;
    render(cur_pos);

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    patrol_update(state);
}
```

Coroutines

Coroutines: functions that can be *suspended* and *resumed*.

Coroutines

Coroutines: functions that can be *suspended* and *resumed*.

Let the compiler generate the state machine for you.

Coroutines

pseudo code

```
void patrol(int& pos, int start, int end)
{
    pos = start;
    auto dir = +1;
    while (true)
    {
        continue return;

        pos += dir;
        if (pos == end)
            dir = -1;
        if (pos == start)
            dir = 1;
    }
}
```

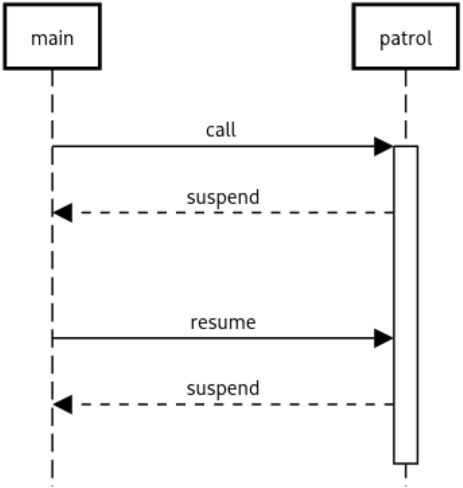
Coroutines

pseudo code

```
auto pos = 0;
auto handle = patrol(pos, 0, 10);
while (true)
{
    auto cur_pos = pos;
    render(cur_pos);

    std::this_thread::sleep_for(std::chrono::seconds(1));
    continue handle;
}
```

Coroutines



Coroutine Execution

Terminology

Terminology

Coroutine

Programmer written version of suspendable function.

- How do I indicate that a function is a coroutine?
- What happens when you call a coroutine?
- What is returned when you call a coroutine?

Terminology

Coroutine state machine

Compiler-generated implementation of suspendable function.

- Programmer can suspend coroutine.
- Programmer can suspend coroutine awaiting something.
- Programmer can exit coroutine.

- What is the initial state?
- What is the final state?
- How much control do you have?

Terminology

Coroutine handle

Interface for controlling a coroutine.

- Programmer can query whether the coroutine is done.
- Programmer can resume execution of the coroutine.
- (Programmer can preemptively destroy the coroutine.)

- Is there a single handle for all coroutines or are there different types?
- Can you implement the handle directly without going through a coroutine?

Terminology

Awaitable

Entity that can be awaited inside a coroutine.

- What can be awaited?
- How do I write my own awaitables?

Terminology

Resumer

Entity that resumes a coroutine when it can make progress.

- Who is responsible for resuming the coroutine? The caller? The awaitable?
- How do I resume a coroutine and how do I know when it's ready?

Coroutines in C++: Basics

Coroutines in C++: Basics

- Coroutine: function that uses `co_await` or `co_return` (or `co_yield`)
- Coroutine state machine: controlled by user-defined *promise*
- Coroutine handle: `std::coroutine_handle`

Defining a coroutine

```
task<int> my_coroutine()
{
    std::puts("my coroutine");
    co_return 42;
}
```

Defining a coroutine

```
task<int> my_coroutine()
{
    std::puts("my coroutine");
    co_return 42;
}
```

- regular function signature; coroutine is implementation detail
- specified return type does not match `co_return` type, instead a “fancy type” is returned

The promise

Simplified

```
struct Promise
{
    // Construct the type returned by the coroutine function.
    ReturnType get_return_object();

    // Controls whether the coroutine suspends initially or after return.
    ??? initial_suspend();
    ??? final_suspend();

    // Called when an exception wants to escape the coroutine.
    void unhandled_exception();

    // Called by `co_return value`.
    void return_value(T&& value);
};
```

std::coroutine_handle

```cpp
template <typename Promise = void> // void means type-erased
struct std::coroutine_handle
{
    static coroutine_handle from_promise(Promise& promise);

    bool done() const;
    void resume() const;
    void destroy() const;

    Promise& promise() const;
};
```

std::coroutine_handle

```
template <typename Promise = void> // void means type-erased
struct std::coroutine_handle
{
    static coroutine_handle from_promise(Promise& promise);

    bool done() const;
    void resume() const;
    void destroy() const;

    Promise& promise() const;
};
```

```
template <typename Promise>
class unique_coro { … };
```

Implementing task

```cpp
template <typename T>
class task
{
public:
    struct promise_type;

    void resume();
    std::optional<T> result() const;

private:
    unique_coro<promise_type> _coro;
};
```

Implementing task: promise_type

```
template <typename T>
struct task<T>::promise_type
{
    std::optional<T> result;

    task get_return_object()
    {
        return {std::coroutine_handle<promise_type>::from_promise(*this)};
    }

    void return_value(T&& value) { result.emplace(std::move(value)); }

    std::suspend_always initial_suspend() noexcept { return {}; }
    std::suspend_always final_suspend()   noexcept { return {}; }
};
```

Compiler transformation

pseudo code

```
task<int> my_coroutine()
{



    std::puts("my coroutine");
    co_return 42;


}
```

Compiler transformation

pseudo code

```
task<int> my_coroutine()
{
    task<int>::promise_type promise = …;
    continue return promise.get_return_object(); // initial suspend

    std::puts("my coroutine");
    promise.return_value(42);

    continue return; // final suspend
}
```

Implementing task: user interface

```cpp
template <typename T>
class task
{
    unique_coro<promise_type> _coro;

public:
    void resume()
    {
        _coro->resume();
    }
    std::optional<T> result() const
    {
        return _coro->promise().result;
    }
};
```

Calling the coroutine

6E8josPWb

```
task<int> t = my_coroutine();
std::printf("result: %d\n", t.result().value_or(-1));
```

Calling the coroutine

6E8josPWb

```
task<int> t = my_coroutine();
t.resume();
std::printf("result: %d\n", t.result().value_or(-1));
```

Calling the coroutine

6E8josPWb

```
task<int> t = my_coroutine();
t.resume();
t.resume();
std::printf("result: %d\n", t.result().value_or(-1));
```

Suspending a coroutine

4c6WfadYv

`continue return` is spelled `co_await std::suspend_always{}`.

Suspending a coroutine

4c6WfadYv

`continue return` is spelled `co_await std::suspend_always{}`.

```
task<int> my_coroutine()
{
    std::puts("my coroutine");
    co_await std::suspend_always{};
    std::puts("after suspend");
    co_return 42;
}
```

```
task<int> t = my_coroutine();
t.resume();
t.resume();
std::printf("result: %d\n", t.result().value_or(-1));
```

Guard Patrol: C++ coroutine

```
task<void> patrol(int& pos, int start, int end)
{
    pos = start;

    auto dir = +1;
    while (true)
    {
        co_await std::suspend_always{};

        pos += dir;
        if (pos == end)
            dir = -1;
        if (pos == start)
            dir = 1;
    }
}
```

E7PTaj5qo

Guard Patrol: C++ coroutine

E7PTaj5qo

```
auto pos = 0;
task<void> t = patrol(pos, 0, 10);
t.resume(); // initial suspend
while (true)
{
    auto cur_pos = pos;
    render(cur_pos);

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    t.resume();
}
```

Coroutines in Rust: Basics

Coroutines in Rust: Basics

- Coroutine: `async fn`
- Coroutine state machine: hand-written or compiler generated
- Coroutine handle: type that implements the `Future` trait

Defining a coroutine

```rust
async fn my_coroutine() -> i32 {
    println!("my coroutine");
    42 // Rust has implicit return
}
```

Defining a coroutine

```
async fn my_coroutine() -> i32 {
    println!("my coroutine");
    42 // Rust has implicit return
}
```

- special function marker; coroutine is not implementation detail
- specified return type matches expression that is returned

The Future trait

```
enum Poll<T> {
    Ready(T),
    Pending
}

trait Future {
    type Output;

    // done() + resume() in one call
    fn poll(self : Pin<&mut Self>, ctx : &mut Context<'_>)
          -> Poll<Self::Output>;
}
```

Compiler transformation

```rust
async fn my_coroutine() -> i32 {
    println!("my coroutine");
    42
}
```

Compiler transformation

```
struct MyCoroutineFuture {}

impl Future for MyCoroutineFuture {
    type Output = i32;

    fn poll(self : Pin<&mut Self>, _ : &mut Context<'_>) -> Poll<i32> {
        println!("my coroutine");
        Poll::Ready(42)
    }
}

fn my_coroutine() -> MyCoroutineFuture {
    MyCoroutineFuture{}
}
```

Calling the coroutine

fETh5E19j

```
let mut fut = my_coroutine();
```

Calling the coroutine

fETh5E19j

```
let mut fut = my_coroutine();
let result = resume(&mut fut);
println!("result: {}", result.unwrap());
```

```
// Just calls `F::poll()`.
fn resume<F: Future>(f: &mut F) -> Option<F::Output>;
```

Suspending a coroutine

z9xGca3rY

`continue return` is spelled `suspend_always().await`

Suspending a coroutine

z9xGca3rY

`continue return` is spelled `suspend_always().await`

```
async fn my_coroutine() -> i32 {
    println!("my coroutine");
    suspend_always().await;
    println!("after suspend");
    42
}
```

```
pub fn main() {
    let mut fut = my_coroutine();
    resume(&mut fut);
    let result = resume(&mut fut);
    println!("result: {}", result.unwrap());
}
```

Implementation of suspend_always()

incomplete

```
struct SuspendAlways { first: bool }

impl Future for SuspendAlways {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, ctx: &mut Context<'_>) -> Poll<()> {
        if replace(self.first, false) {
            Poll::Pending
        } else {
            Poll::Ready(())
        }
    }
}

fn suspend_always() -> SuspendAlways {
    SuspendAlways{ first: true }
}
```

Guard Patrol: Rust coroutine

zqddYY1EE

```rust
async fn patrol(pos : &Cell<i32>, start : i32, end : i32) {
    pos.set(start);

    let mut dir = 1;
    loop {
        suspend_always().await;

        pos.set(pos.get() + dir);
        if pos.get() == end {
            dir = -1;
        }
        if pos.get() == start {
            dir = 1;
        }
    }
}
```

Guard Patrol: Rust coroutine

zqddYY1EE

```
let pos = Cell::new(0);
let mut fut = patrol(&pos, 0, 10);
resume(&mut fut); // initially suspended
loop {
    render(pos.get());

    std::thread::sleep(std::time::Duration::from_millis(100));
    resume(&mut fut);
}
```

Basic coroutines: comparison

Basic coroutines: comparison

C++

- coroutine is an implementation detail
- written return type matches result from call, does not match returned expression
- promise type allows customization of the generated state machine
- `std::coroutine_handle` can only be used with coroutines

Rust

- coroutine-ness is visible in the interface
- written return type matches returned expression, does not match result from call
- state machine can either be generated or hand-written
- `Future` trait can be implemented yourself

Coroutines in C++: Awaitables and Resumer

Coroutines in C++: Awaitables and Resumer

- Awaitable: something with overloaded `operator co_await` (or `Awaiter` itself)
- Resumer: the awaitable

Awaitable

Simplified

```cpp
struct Awaitable
{
    // optional: Awaitable can be an Awaiter itself
    Awaiter operator co_await();
};
```

```cpp
puts("before await");
auto value = co_await awaitable;
puts("after await");
```

Awaiter

Simplified

```
struct Awaiter
{
    bool await_ready();
    void await_suspend(std::coroutine_handle<Promise> suspended_coroutine);
    T await_resume();
};
```

Awaiter

Simplified

```
struct Awaiter
{
    bool await_ready();
    void await_suspend(std::coroutine_handle<Promise> suspended_coroutine);
    T await_resume();
};
```

```
puts("before await");
auto awaiter = awaitable.operator co_await();
if (!awaiter.await_ready())
{
    awaiter.await_suspend(current_coroutine_handle);
    continue return;
}
auto value = awaiter.await_resume();
puts("after await");
```

std::suspend_always

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

    void await_suspend(std::coroutine_handle<>)
    {}

    void await_resume() {}
};
```

initial_suspend() and final_suspend()

`Promise::initial_suspend()` and `Promise::final_suspend()` return awaitables.

```
task<int> my_coroutine()
{
    task<int>::promise_type promise = …;
    co_await promise.initial_suspend();

    …

    co_await promise.final_suspend();
}
```

Coroutine resumer

It is the responsibility of `await_suspend()` to schedule the coroutine for resumption.

Coroutine resumer

It is the responsibility of `await_suspend()` to schedule the coroutine for resumption.

```
task<void> patrol(int& pos, int start, int end)
{
    …

    while (true)
    {
        co_await std::suspend_always{}; // bad

        …
    }
}
```

Introducing a timer

Efhhb9MTE

```
class timer
{
    struct awaiter;

public:
    // Suspend coroutine until next tick.
    awaiter operator co_await();

    // Resume all coroutines waiting for the timer.
    void tick();
};
```

Using a timer

Efhhb9MTE

```
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await tmr;
    …
}
```

```
timer tmr;
task<void> t = patrol(tmr, pos, 0, 10);
t.start(); // renamed from .resume()
while (true)
{
    …
    std::this_thread::sleep_for(std::chrono::seconds(1));
    tmr.tick();
}
```

Implementing a timer

Efhhb9MTE

Insight: the Awaiter is stored as part of the coroutine.

```
class timer
{
    struct awaiter
    {
        timer* tmr;
        std::coroutine_handle<> waiting;
        awaiter* next;

        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<> suspended);
        void await_resume() {}
    };
    awaiter* _head;
};
```

Implementing a timer

Efhhb9MTE

```
void awaiter::await_suspend(std::coroutine_handle<> suspended)
{
    waiting = suspended;
    next = std::exchange(tmr->_head, this);
}
```

```
void timer::tick()
{
    auto cur = std::exchange(_head, nullptr); // important!
    while (cur)
    {
        auto next = cur->next;
        cur->waiting.resume();
        cur = next;
    }
}
```

Using a timer

Efhhb9MTE

```
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await tmr;
    …
}
```

```
timer tmr;
task<void> t = patrol(tmr, pos, 0, 10);
t.start(); // renamed from .resume()
while (true)
{
    …
    std::this_thread::sleep_for(std::chrono::seconds(1));
    tmr.tick();
}
```

Chaining coroutines

```
task<void> move(timer& tmr, int& pos, int steps, int dir)
{
    for (auto i = 0; i != steps; ++i)
    {
        co_await tmr;
        pos += dir;
    }
}

task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    move(tmr, pos, 1, dir);
    …
}
```

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir)
{
    for (auto i = 0; i != steps; ++i)
    {
        co_await tmr;
        pos += dir;
    }
}

task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

Execute continuation in promise

```
struct task<T>::promise_type
{
    std::coroutine_handle<> continuation;

    auto final_suspend() noexcept
    {
        struct awaiter
        {
            void await_suspend(std::coroutine_handle<promise_type> suspended)
            {
                if (suspended.promise().continuation)
                    suspended.promise().continuation.resume();
            }
        };
        return awaiter{};
    }
```

84oPjvPTa

Set continuation in co_await

84oPjvPTa

```
auto task<T>::operator co_await() const
{
    struct awaiter
    {
        std::coroutine_handle<promise_type> handle;

        void await_suspend(std::coroutine_handle<> suspended)
        {
            handle.promise().continuation = suspended;
            handle.resume();
        }
    };
    return awaiter{*_coro};
}
```

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir);
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

1 Resume execution of `patrol()`.

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir);
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

1. Resume execution of `patrol()`.
2. Call `move()`, which does nothing.

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir);
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

1. Resume execution of `patrol()`.
2. Call `move()`, which does nothing.
3. Call `task<void>::operator co_await`: suspend `patrol()`, set continuation of `move()` to `patrol()`, resume `move()`

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir);
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

1. Resume execution of `patrol()`.
2. Call `move()`, which does nothing.
3. Call `task<void>::operator co_await`: suspend `patrol()`, set continuation of `move()` to `patrol()`, resume `move()`
4. Call `timer::operator co_await`: suspend `move()`, register `move()` to be continued on `tick()`

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir);
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

1. Resume execution of `patrol()`.
2. Call `move()`, which does nothing.
3. Call `task<void>::operator co_await`: suspend `patrol()`, set continuation of `move()` to `patrol()`, resume `move()`
4. Call `timer::operator co_await`: suspend `move()`, register `move()` to be continued on `tick()`
5. `timer::tick()` resumes `move()`, which finishes

Chaining coroutines

84oPjvPTa

```
task<void> move(timer& tmr, int& pos, int steps, int dir);
task<void> patrol(timer& tmr, int& pos, int start, int end)
{
    …
    co_await move(tmr, pos, 1, dir);
    …
}
```

1. Resume execution of `patrol()`.
2. Call `move()`, which does nothing.
3. Call `task<void>::operator co_await`: suspend `patrol()`, set continuation of `move()` to `patrol()`, resume `move()`
4. Call `timer::operator co_await`: suspend `move()`, register `move()` to be continued on `tick()`
5. `timer::tick()` resumes `move()`, which finishes
6. `task<void>::final_suspend` resumes continuation (i.e. `patrol()`)

Many nested calls

h16o8rc5q

```
task<void> coro_b()
{
    co_return;
}

task<void> coro_a()
{
    for (auto i = 0; i != 1000000; ++i)
        co_await coro_b();
}
```

Many nested calls

h16o8rc5q

```
task<void> coro_b()
{
    co_return;
}

task<void> coro_a()
{
    for (auto i = 0; i != 1000000; ++i)
        co_await coro_b();
}
```

Segmentation fault.

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`

Many nested calls

Call stack (growing down):

- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`
- `task::awaiter::await_suspend(coro_b)`
- `coroutine_handle::resume(coro_b)`
- `coroutine_handle::resume(coro_a)`
- ...

Tail calls

haY64vPfj

```
// task's awaiter
void await_suspend(std::coroutine_handle<> suspended)
{
    handle.promise().continuation = suspended;
    handle.resume();
}
```

```
// final_suspend()'s awaiter
void await_suspend(std::coroutine_handle<promise_type> suspended)
{
    if (suspended.promise().continuation)
        suspended.promise().continuation.resume();
}
```

Tail calls

haY64vPfj

```
// task's awaiter
auto await_suspend(std::coroutine_handle<> suspended)
{
    handle.promise().continuation = suspended;
    return handle;
}
```

```
// final_suspend()'s awaiter
auto await_suspend(std::coroutine_handle<promise_type> suspended)
{
    if (suspended.promise().continuation)
        return suspended.promise().continuation;
    else
        return std::noop_coroutine();
}
```

Coroutines in Rust: Awaitables and Resumer

Coroutines in Rust: Awaitables and Resumer

- Awaitable: a type that implements `Future`
- Resumer: some user-written executor

Coroutines in Rust: Awaitables and Resumer

- Awaitable: a type that implements `Future`
- Resumer: some user-written executor

Question: how is the executor informed that it should call `poll()` again?

Context and Waker

Simplified

`Context`: provides a `Waker`

```
struct Context { … };

impl Context {
    fn waker(&self) -> &Waker;
}
```

`Waker`: called by a `Future` when it is ready to be polled again

```
struct Waker { … };

impl Waker {
    fn wake(self);
    fn wake_by_ref(&self);
}
```

Complete Implementation of suspend_always()

Kv3GnecT7

```rust
struct SuspendAlways {
    first: bool
}

impl Future for SuspendAlways {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, ctx: &mut Context<'_>) -> Poll<()> {
        if replace(self.first, false) {
            ctx.waker().wake_by_ref();
            Poll::Pending
        } else {
            Poll::Ready(())
        }
    }
}
```

Writing a resumer

z9xGca3rY

```
fn resume<F: Future>(f: &mut F) -> Option<F::Output> {
    let waker = get_noop_waker();
    let mut ctx = Context::from_waker(&waker);

    match F::poll(unsafe { Pin::new_unchecked(f) }, &mut ctx) {
        Poll::Pending => None,
        Poll::Ready(val) => Some(val),
    }
}
```

```
let mut fut = my_coroutine();
resume(&mut fut);
```

Introducing a timer

7G6WY6Tc9

```rust
struct Timer {
    time: i32,
    waker: Option<Waker>
}

impl Timer {
    fn new() -> RefCell<Timer> { … }

    fn block(self_ : &RefCell<Timer>) -> TimerFuture { … }

    fn tick(self_ : &RefCell<Timer>) { … }
}
```

Using a timer

7G6WY6Tc9

```
async fn patrol(timer: &RefCell<Timer>, pos: &Cell<i32>, start: i32, end: i32) {
    …
    Timer::block(&timer).await;
    …
}
```

```
let tmr = Timer::new();
let mut fut = patrol(&tmr, &pos, 0, 10);
resume(&mut fut); // initially suspended
loop {
    …
    std::thread::sleep(std::time::Duration::from_millis(100));
    Timer::tick(&tmr);
    resume(&mut fut);
}
```

Implementing a timer

7G6WY6Tc9

```rust
struct TimerFuture<'a> {
    cur_time: i32,
    timer: &'a RefCell<Timer>
}

impl Future for TimerFuture<'_> {
    type Output = ();
    fn poll(self: Pin<&mut Self>, ctx: &mut Context<'_>) -> Poll<()> {
        if self.cur_time == self.timer.borrow().time {
            self.timer.borrow_mut().waker = Some(ctx.waker().clone());
            Poll::Pending
        } else {
            Poll::Ready(())
        }
    }
}
```

Implementing a timer

7G6WY6Tc9

```
impl Timer {
    fn tick(self_ : &RefCell<Timer>) {
        let mut self_ = self_.borrow_mut();
        self_.time += 1;
        if let Some(waker) = self_.waker.take() {
            waker.wake();
        }
    }
}
```

Chaining coroutines

3rz3db3E9

```rust
async fn move_(timer: &RefCell<Timer>, pos: &Cell<i32>, steps: i32, dir: i32) {
    for _ in 0..steps {
        Timer::block(&timer).await;
        pos.set(pos.get() + dir);
    }
}

async fn patrol(timer: &RefCell<Timer>, pos: &Cell<i32>, start: i32, end: i32) {
    …
    loop {
        move_(&timer, &pos, 1, dir).await;
        …
    }
}
```

Awaitables and resumer: comparison

Awaitables and resumer: comparison

C++

- Awaitable is user-defined type with `operator co_await`
- `await_suspend()` schedules the coroutine for resumption (bottom up)
- coroutine chaining requires library code, `co_await` implementation
- you can customize what happens on resumption
- coroutine is only resumed when it can definitely make progress

Rust

- Awaitable is type implementing `Future` trait
- you need to write something that executes the top-level future (top down)
- coroutine chaining is part of the language
- you can write the entire state machine yourself
- coroutine can be polled unnecessarily, `Waker` used to notify when polling should be done

C++: Executing a coroutine on a thread pool

```cpp
class thread_pool
{
public:
    awaitable schedule() const;
};
```

```cpp
task<void> my_coroutine(const thread_pool& pool)
{
    std::puts("hello from main thread");
    co_await pool.schedule();
    std::puts("hello from thread pool");
}
```

Rust: Executing a coroutine on a thread pool

```rust
struct ThreadPool { … };

impl ThreadPool {
    fn spawn<F: Future>(f: F);
}
```

```rust
async fn my_coroutine();

…

pool.spawn(my_coroutine());
```

Sockets

C++:

```
auto buffer = co_await socket.read();
```

Rust:

```
let buffer = socket.read().await;
```

Rust: Sockets

```rust
struct SocketRead<'a> {
    socket: &'a Socket,
}
impl Future for SocketRead<'_> {
    type Output = Vec<u8>;
    fn poll(self : Pin<&mut Self>, ctx : &mut Context<'_>)
      -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.do_sync_read())
        } else {
            self.socket.on_data(ctx.waker());
            Poll::Pending
        }
    }
}
```

C++: Sockets

```
struct socket_read
{
    socket* s;
    bool await_ready()
    {
        return s->has_data_to_read();
    }
    void await_suspend(std::coroutine_handle<> waiting)
    {
        s->on_data([waiting] { waiting.resume(); });
    }
    std::vector<char> await_resume()
    {
        return s->do_sync_read();
    }
};
```

C++: Sockets

Only notify that it is ready, but don't resume yet:

```
void socket_read::await_suspend(std::coroutine_handle<> waiting)
{
    s->on_data([s, waiting] { s->context.can_resume(waiting); });
}
```

Coroutine Allocation

Coroutines store state

pseudo code

```
void my_coroutine()
{
    std::string str;
    std::cin >> str;
    continue return;
    std::cout << str << '\n';
}

int main()
{
    auto handle = my_coroutine();
    continue handle;
}
```

Stackful coroutines

Idea: Each coroutine uses a separate stack.

Stackful coroutines

Idea: Each coroutine uses a separate stack.

Problem: How big of a stack? What about wasted space? Stack overflow?

Stackful coroutines

Idea: Each coroutine uses a separate stack.

Problem: How big of a stack? What about wasted space? Stack overflow?

```
void my_coroutine()
{
    continue return;
}
void normal_function(void (*f)())
{
    f();
}
int main()
{
    auto handle = normal_function(&my_coroutine);
}
```

Rust: Allocation

Idea: store data in `Future` type.

```
struct MyCoroutineFuture
{
    state: i32,
    foo: i32,
    bar: i32
}

impl Future for MyCoroutineFuture { … }

fn my_coroutine() -> MyCoroutineFuture {
    MyCoroutineFuture{ state: 0, foo: 0, bar: 42 }
}
```

C++: Allocation

Idea: heap allocate coroutine state.

C++: Allocation

Idea: heap allocate coroutine state.

Coroutine state:

- promise object
- parameters
- current suspension point (state machine state)
- local variables that need to persist between suspensions

`std::coroutine_handle` is a type-erased pointer to that state.

Comparison

Rust: coroutine state part of type system

- you can figure out the type of the coroutine state
- coroutine state can be treated like any variable and put of the stack
- size of state can be queried at compile-time
- type has to be determined before optimizations happen and is part of the ABI
- coroutine state bigger than necessary

C++: coroutine state type-erased

- you cannot figure out the type of the coroutine state
- coroutine state has to be heap allocated as it is type-erased
- size of state cannot be queried at compile-time
- type can be determined after optimizations happen and is not part of the ABI
- coroutine state only as big as necessary

HALO: Heap allocation elision optimization

```
task<int> my_coroutine();

int main()
{
    auto state = allocate_coroutine_state(&my_coroutine);
    auto task = state->promise->get_return_object();
    …
    // .destroy() calls deallocate_coroutine_state(state)
}
```

HALO: Heap allocation elision optimization

```
task<int> my_coroutine();

int main()
{
    auto memory = alloca(coroutine_state_size);
    auto state = construct_coroutine_state(memory, &my_coroutine);
    auto task = state->promise->get_return_object();
    …
}
```

Conclusion

Conclusion

Coroutines: functions that can be *suspended* and *resumed*.

- C++ gives you some control over state machine, Awaitables take care of resumption
- Rust allows you to write entire state machine by hand, top-level executors take care of resumption after notification
- trade-off: typed vs type-erased coroutine state

jonathanmueller.dev/talk/accu2022

Twitter: @foonathan

Support me: jonathanmueller.dev/support-me/