Coroutines Customization

(Getting familiar with C++)

/mohamed-ayoub-akkaoui

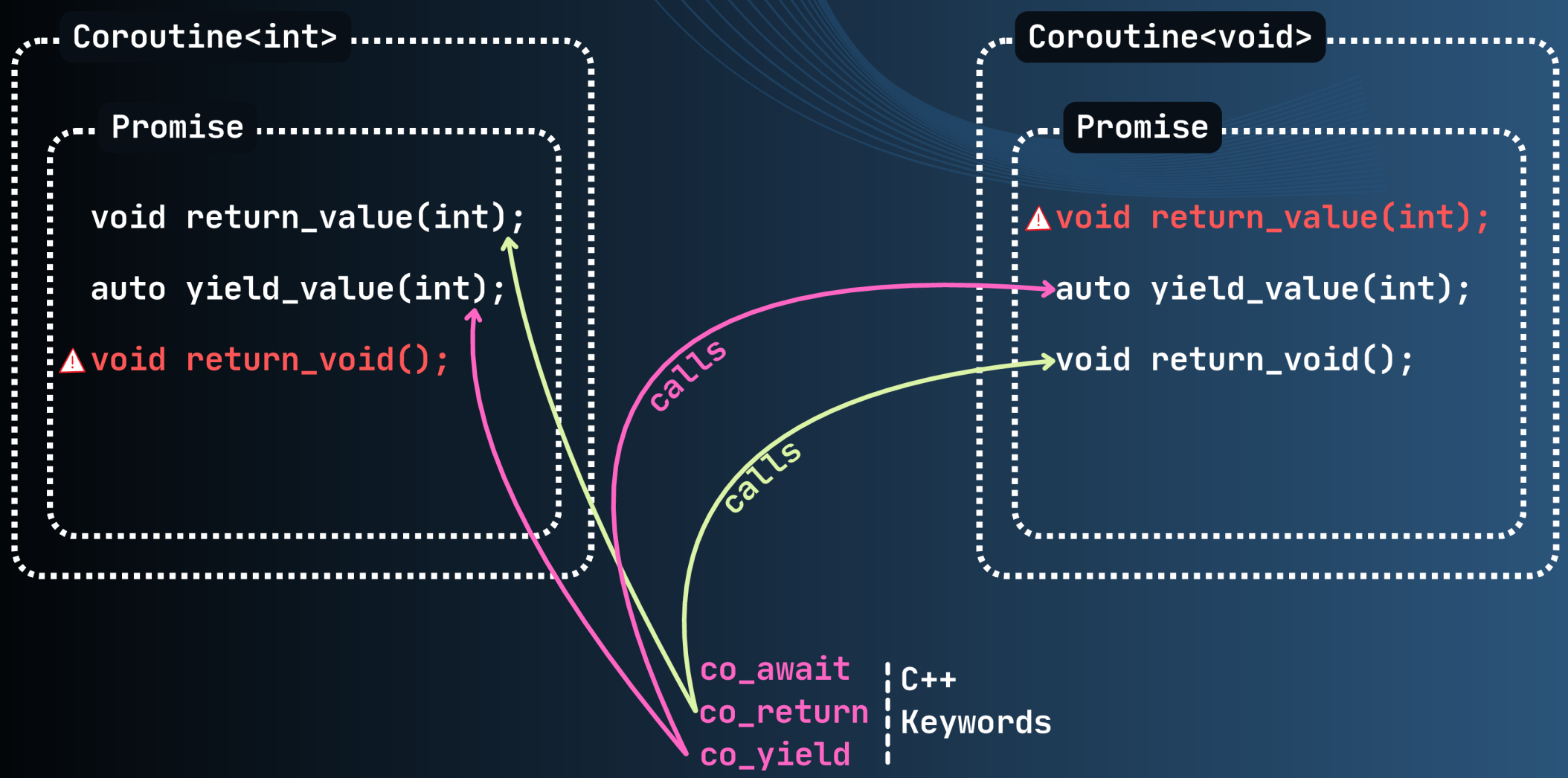
1 Awaitable Type

> The C++ standard library defines two types of Awaitable Types; std::suspend_always and std::suspend_never. Fortunately, with current compilers, you can define your own type by implementing the three following methods: await_ready, await_suspend and await_resume.

```cpp
#include<coroutine>

class Awaitable {

    bool await_ready() {
        bool isReady{ false };
        if (/* result or process finished */)
        {
            isReady = true;
        }
        return isReady;
    }

    auto await_suspend( std::coroutine_handle<>  ) {
        /* This function will get called if await_ready returns false */
        /* In the body you can either return an other coroutine_handle type to change the call execution */
        /* Or you can return nothing (void) */
    }

    auto await_resume() {
        /* This function will get called whenever the coroutine is resumed */
        /* In generale it is the final result of the expression 'co_await exo' */
        /* It could return a value or nothing (void) */
    }
};
```

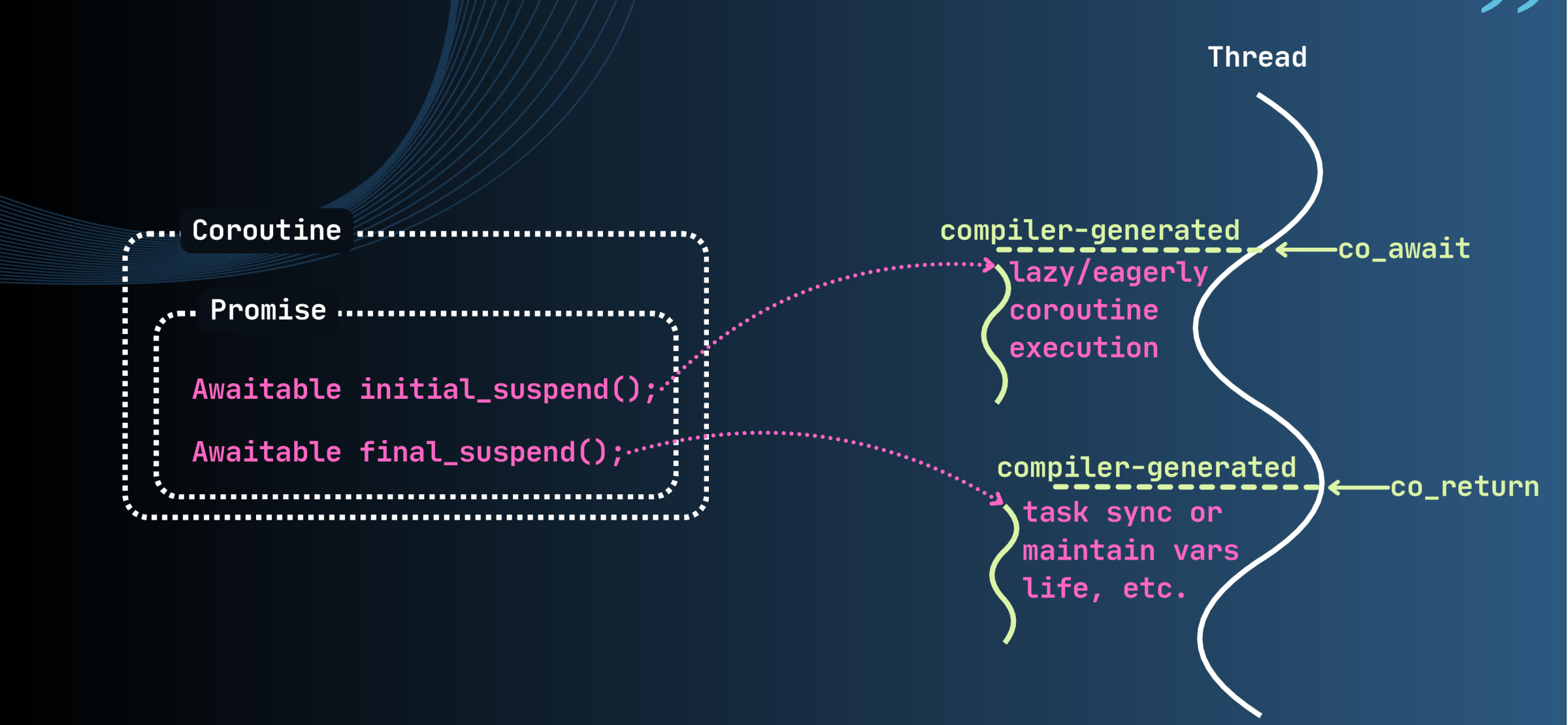
2 Coroutine Return Type

> It would have been great to have a native flexibility on the coroutine return type, however, you can only return a non-void or void type. The main object which dispatches the returned result is the Promise, in it, you can either implement return_value or return_void functions. On the other hand, it is not a blocking point as you can overcome this by using std::variant or by defining a two template specializations for the relevant coroutine.



3 Coroutine Suspend Points

> You can always control your coroutine through co_await, co_yield and co_return; they are the explicit suspend points which can suspend the current coroutine based on your code logic and your needs. The most interesting group is the implicit suspend points which are highly customizable and can have an impact after the coroutine construction or before its destruction; initial_suspend and final_suspend respectively.



```cpp
#include <iostream>
#include <coroutine>
#include <mutex>
#include <thread>

std::mutex global_mutex;

struct Awaitable {
    bool await_ready() const noexcept { return false; } // Always suspends
    void await_suspend(std::coroutine_handle<>) const noexcept {
        std::this_thread::sleep_for(std::chrono::milliseconds(100)); // Simulate some I/O task
    }
    void await_resume() const noexcept { /* Nothing to resume */ }
};

struct Task {
    struct promise_type {
        std::mutex& mtx; // Reference to external mutex for synchronization

        promise_type(std::mutex& m) : mtx(m) {}

        // Implicit suspension: Coroutine suspends before execution starts
        std::suspend_always initial_suspend() {
            std::cout << "Acquiring lock (initial_suspend)\n";
            mtx.lock(); // Lock the shared resource at the start
            return {};
        }

        // Coroutine returns void, so we implement return_void
        void return_void() {
            std::cout << "Task finished.\n";
        }

        // Implicit suspension: Coroutine suspends after completion
        std::suspend_always final_suspend() noexcept {
            std::cout << "Releasing lock (final_suspend)\n";
            mtx.unlock(); // Unlock the shared resource at the end
            return {};
        }

        Task get_return_object() {
            return Task{};
        }
    };

    // Coroutine needs to implement the suspend points
    Task operator co_await() {
        return *this;
    }
};

Task async_task() {
    std::cout << "Starting async task...\n";
    co_await Awaitable(); // Simulating some I/O operation (explicit suspension)
    co_return;
}

int main() {
    // Launching the coroutine in a separate thread
    std::thread t1([] { async_task(); });

    // Simultaneously running another coroutine
    std::thread t2([] { async_task(); });

    t1.join();
    t2.join();
}

```

Have you explored coroutine customization in your projects?

Comment Bellow

x x x x