Introduction in C++20 coroutines

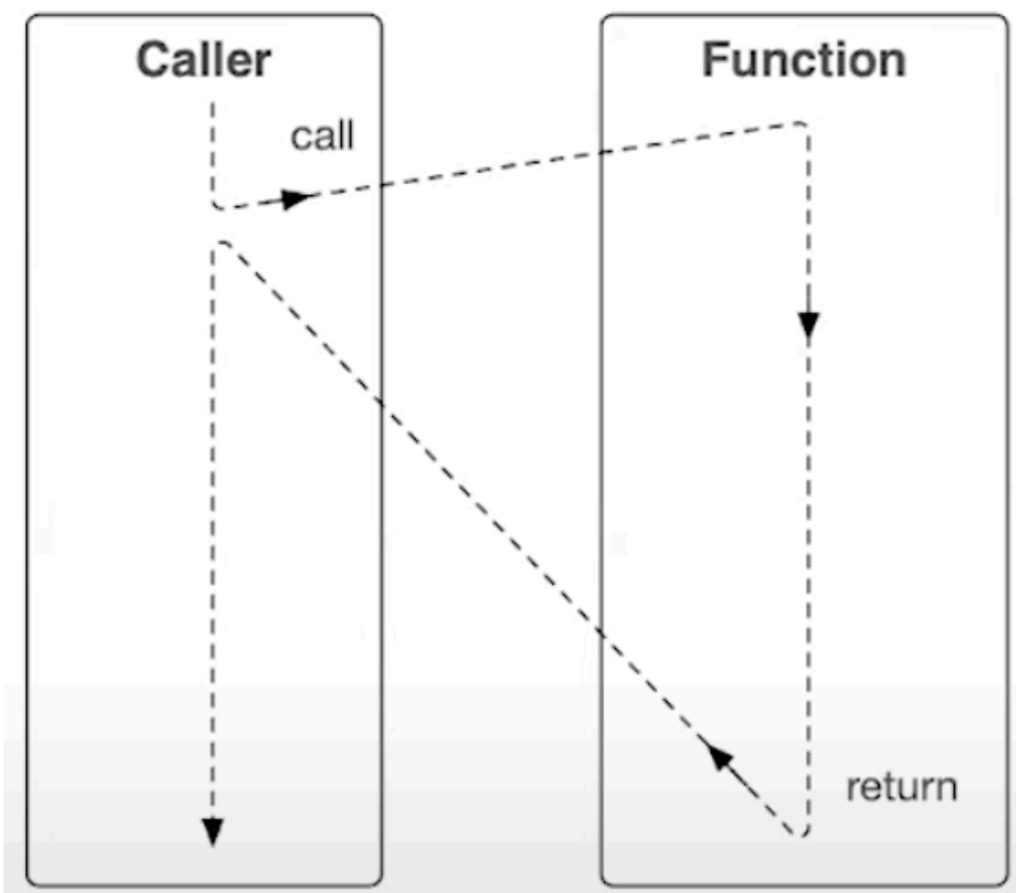
Grinko Kirill

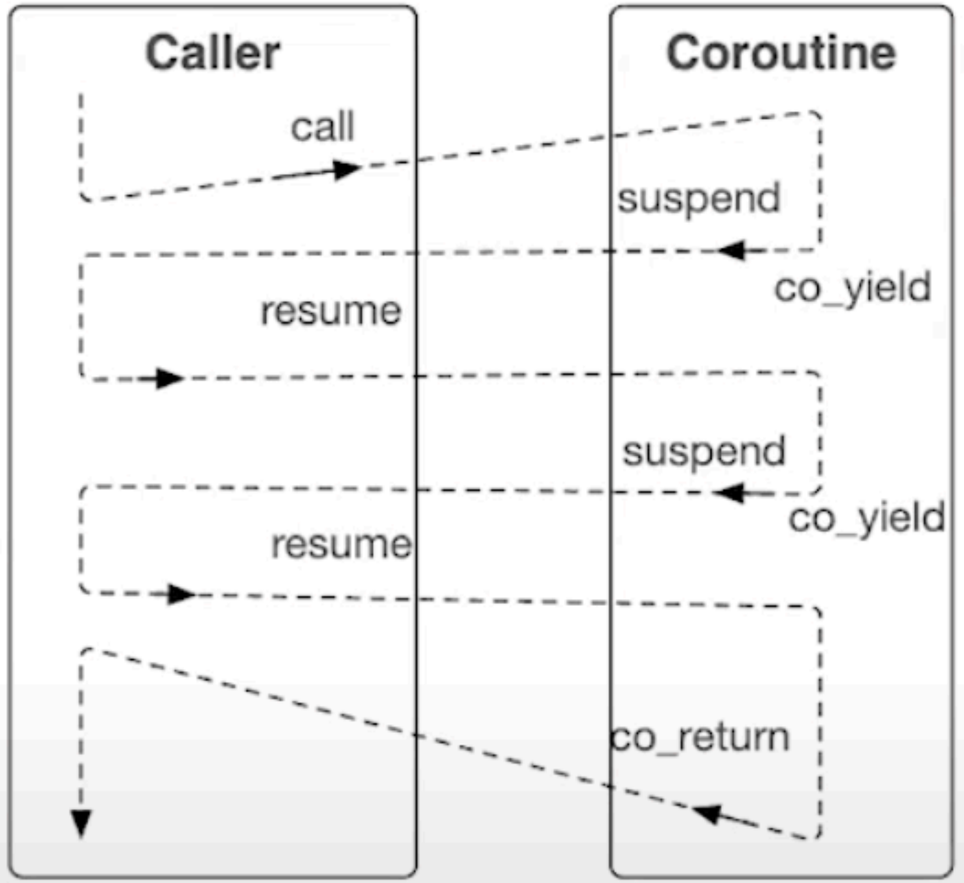
Subroutine vs Coroutine

Сопрограмма (*coroutine*) — это процедура, из вызова которой можно выйти (*остановить* вызов, операция *suspend*), а затем вернуться в этот вызов и продолжить исполнение с точки остановки (*возобновить* вызов, операция *resume*).

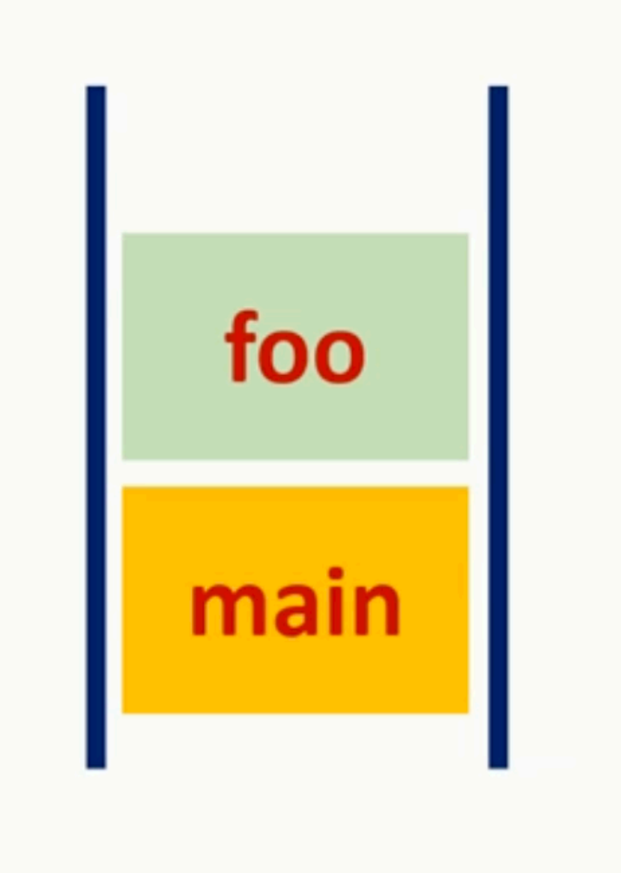
Сопрограмма расширяет понятие *подпрограммы* (*subroutine*), вызов которой нельзя остановить, а можно лишь завершить.

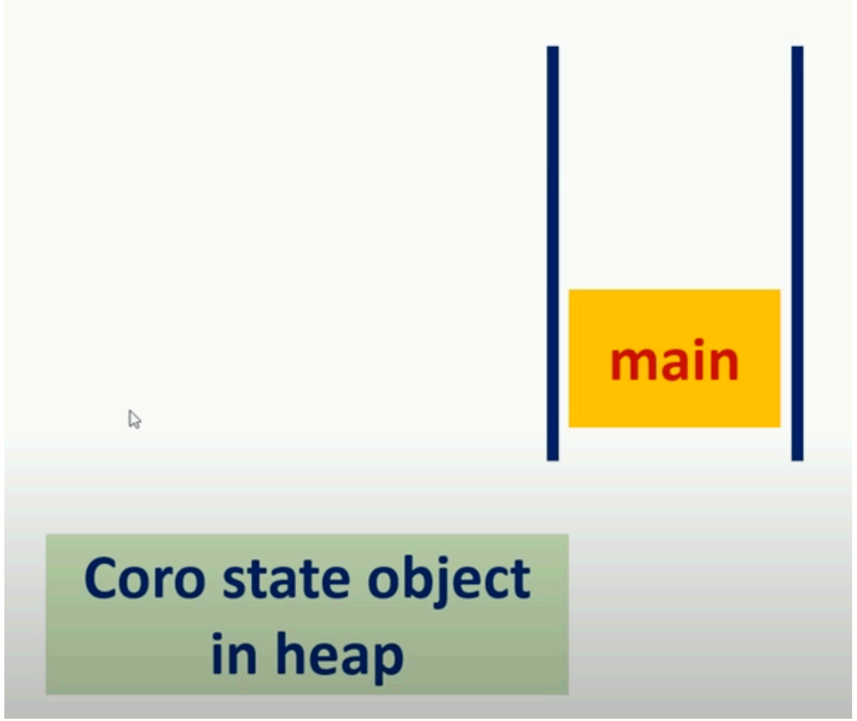
Subroutine vs Coroutine



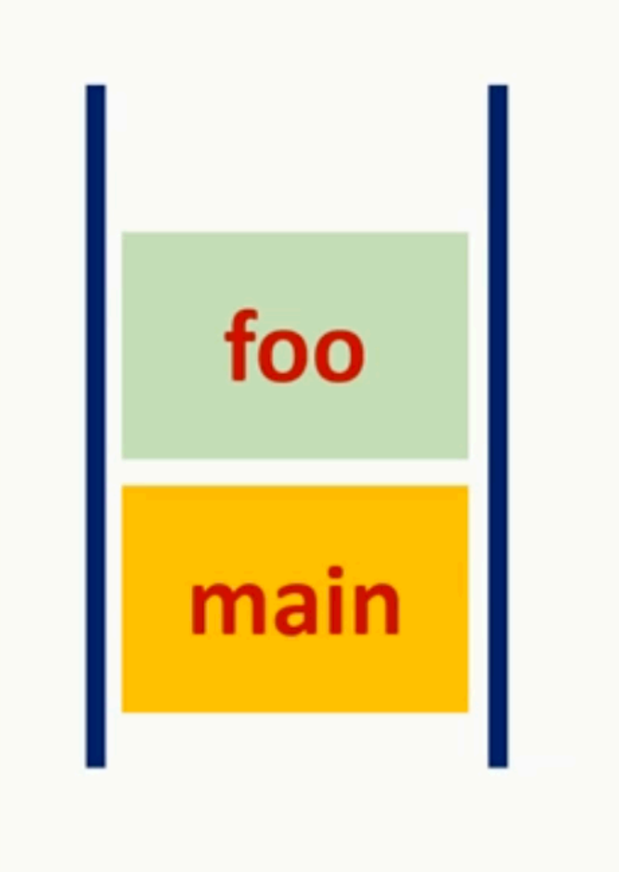


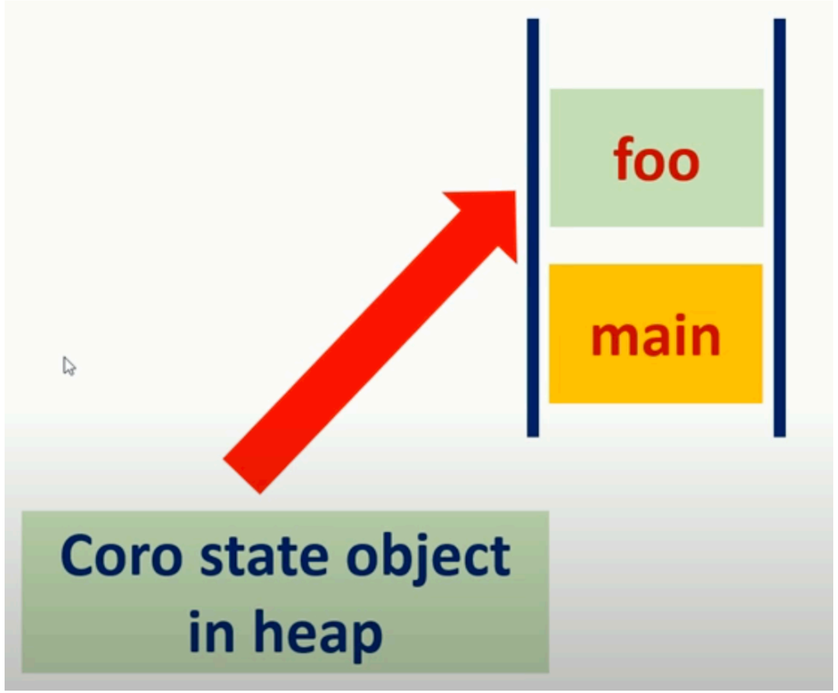
Subroutine vs Stackless coroutine



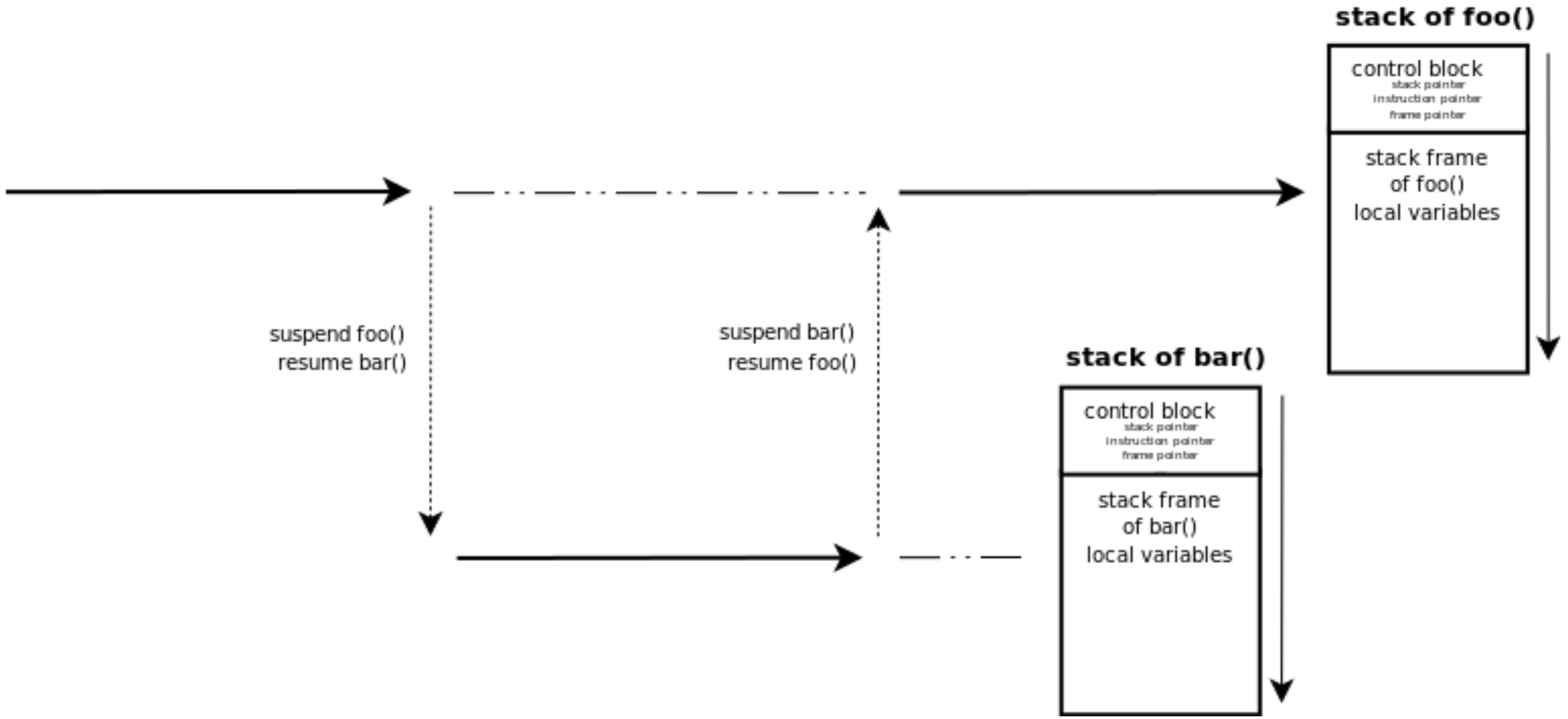


Subroutine vs Stackless coroutine





Stackfull coroutine



Stackfull coroutine

```
30  class Coroutine {
31   public:
32    explicit Coroutine(Body body);
33
34    void Resume();
35    void Suspend();
36
37    bool IsDone() const;
38
39   protected:
40    void Run() noexcept final;
41
42   private:
43    bool is_done_;
44    Body body_;
45    sure::ExecutionContext coro_context_;
46    sure::ExecutionContext outer_context_;
47    sure::stack::GuardedMmapStack stack_;
48  };
49
```

Coroutines in C++20

- co_yield
- co_await
- co_return

Coroutines in C++20

- the `co_await` expression — to suspend execution until resumed

```
task<> tcp_echo_server()
{
    char data[1024];
    while (true)
    {
        std::size_t n = co_await socket.async_read_some(buffer(data));
        co_await async_write(socket, buffer(data, n));
    }
}
```

Coroutines in C++20

- the `co_yield` expression — to suspend execution returning a value

```
generator<unsigned int> iota(unsigned int n = 0)
{
    while (true)
        co_yield n++;
}
```

Coroutines in C++20

- the `co_return` statement — to complete execution returning a value

```
lazy<int> f()
{
    co_return 7;
}
```

Coroutines in C++20

Restrictions

Coroutines cannot use variadic arguments, plain return statements, or placeholder return types (auto or Concept).

Consteval functions, constexpr functions, constructors, destructors, and the main function cannot be coroutines.

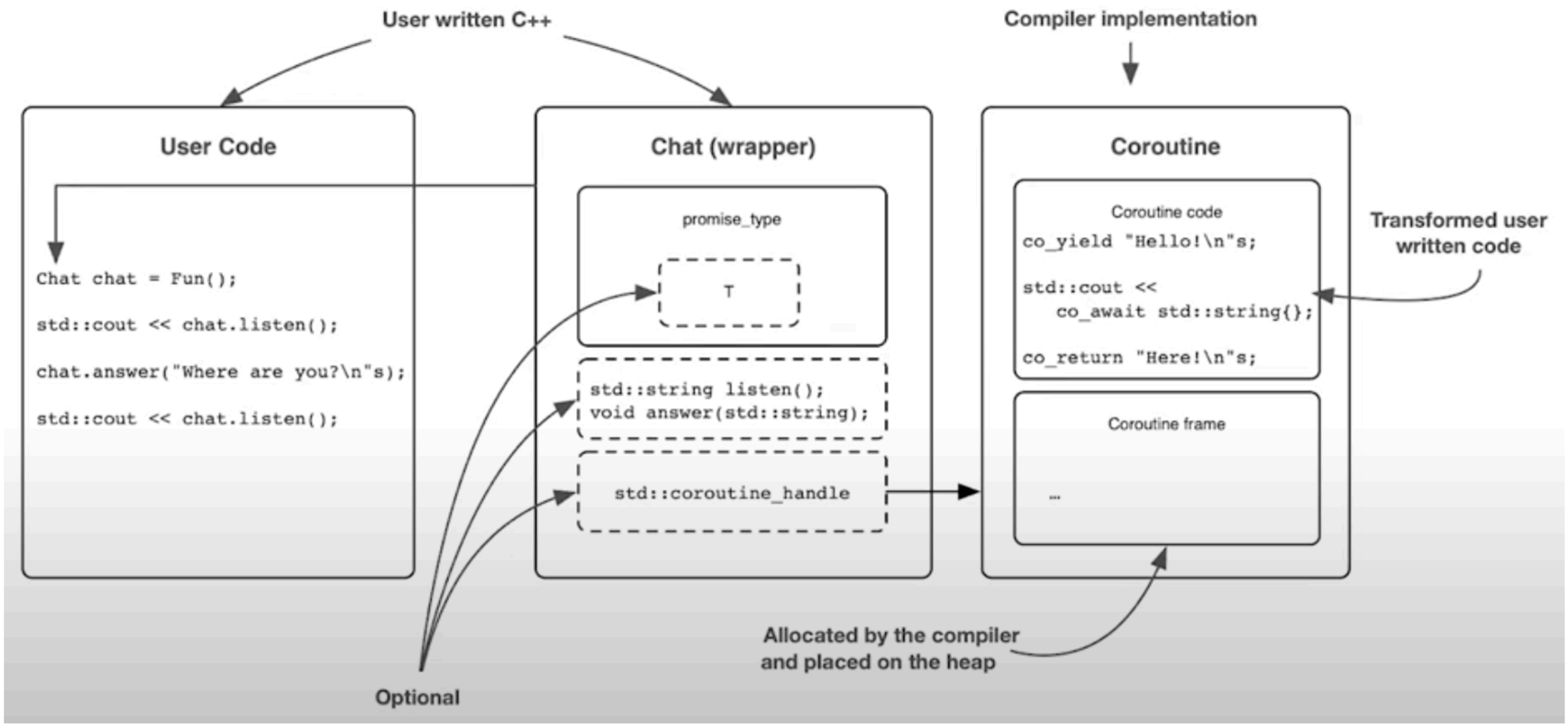
Associated with coroutine

- promise object
- coroutine state
- coroutine handle

Coroutines in C++20

```
{
    Promise promise;
    co_await promise.initial_suspend();
    try {
        <тело функции>
    } catch (...) {
        promise.unhandled_exception();
    }
FinalSuspend:
    co_await promise.final_suspend();
}
```

Coroutines in C++20



Workflow

- Корутина начинает выполнение
  - аллоцирование frame корутины при необходимости.
  - копирование всех параметров функции в frame корутины.
  - создание promise объекта `promise`.
  - вызов `promise.get_return_object()` для создания handle корутины и сохранение такового в локальной переменной. Результат вызова будет возвращен вызывающей стороне при первой приостановке корутины.
  - вызов `promise.initial_suspend()` и ожидание `co_await` результата. Данный тип promise обычно возвращает `suspend_never` для корутин немедленного выполнения или `suspend_always` для ленивых корутин.
  - тело корутины выполняется начинает выполнение после `co_await promise.initial_suspend()`

Workflow

- Корутины достигают точки приостановки
  - возвращаемый объект `promise.get_return_object()` возвращается вызывающей сущности который инициирует продолжение выполнение корутины

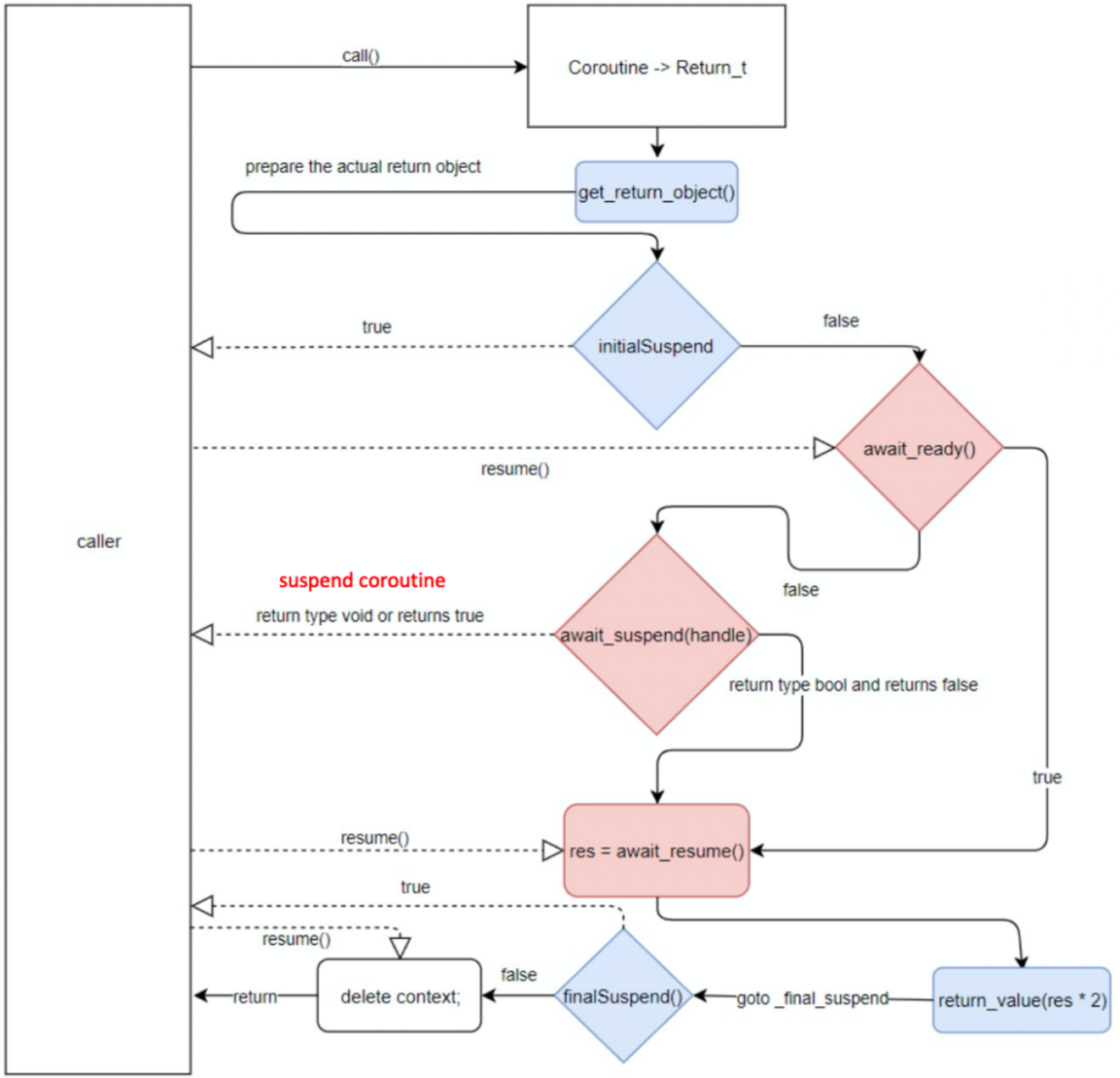
Workflow

- Корутина достигает `co_return`
  - вызывается `promise.return_void()` для `co_return` или `co_return expression`, где `expression` имеет тип `void`
  - вызывается `promise.return_value(expression)` для `co_return expression`, где `expression` имеет тип отличный от `void`
  - удаляется весь стек созданных переменных
  - вызывается `promise.final_suspend()` и ожидается `co_await` результат

Workflow

- Корутина уничтожается (посредством завершения через `co_return`, необработанного исключения или через handle корутины)
  - вызывается деструктор promise объекта
  - вызывается деструктор параметров функции
  - освобождается память используемая frame корутины
  - передача выполнения вызывающей сущности

Workflow


caller
call()
Coroutine -> Return_t
prepare the actual return object
get_return_object()
initialSuspend
true
false
resume()
await_ready()
false
suspend coroutine
return type void or returns true
await_suspend(handle)
return type bool and returns false
true
resume()
res = await_resume()
true
resume()
delete context;
return
false
finalSuspend()
goto _final_suspend
return_value(res * 2)

std::coroutine_handle

Member functions

(constructor)	constructs a `coroutine_handle` object (public member function)
`operator=`	assigns the `coroutine_handle` object (public member function)

Conversion

`operator coroutine_handle<>`	obtains a type-erased `coroutine_handle` (public member function)

Observers

`done`	checks if the coroutine has completed (public member function)
`operator bool`	checks if the handle represents a coroutine (public member function)

Control

`operator()` **`resume`**	resumes execution of the coroutine (public member function)
`destroy`	destroys a coroutine (public member function)

Promise Access

`promise`	access the promise of a coroutine (public member function)
`from_promise` [static]	creates a `coroutine_handle` from the promise object of a coroutine (public static member function)

Export/Import

`address`	exports the underlying address, i.e. the pointer backing the coroutine (public member function)
`from_address` [static]	imports a coroutine from a pointer (public static member function)

Non-member functions

`operator==` **`operator<=>`** (C++20)	compares two `coroutine_handle` objects (function)

Helper classes

`std::hash``<std::coroutine_handle>` (C++20)	hash support for **`std::coroutine_handle`** (class template specialization)

Task & Generator

- Task: A coroutine that does a job without returning a value.
- Generator: A coroutine that does a job and returns a value (either by co_return or co_yield).

Eager(greedy) evaluation

```
3   std::vector<int> FibonacciEager(size_t count) {
4     std::vector<int> result;
5     result.reserve(count);
6
7     int a = 0;
8     int b = 1;
9     for (size_t i = 0; i < count; ++i) {
10      result.push_back(a);
11      int next = a + b;
12      a = b;
13      b = next;
14    }
15
16    return result;
17  }
```

Lazy evaluation

```
 5    Generator<int> FibonacciLazy() {
 6      int a = 0;
 7      int b = 1;
 8      while (true) {
 9        co_yield a;
10        int next = a + b;
11        a = b;
12        b = next;
13      }
14    }
15
```

C++23 std::generator

std::generator

Defined in header <generator>

```
template<
    class Ref,
    class V = void,
    class Allocator = void >
class generator
    : public ranges::view_interface<generator<Ref, V, Allocator>>
```

(1) (since C++23)

```
namespace pmr {
    template< class Ref, class V = void >
    using generator =
        std::generator<Ref, V, std::pmr::polymorphic_allocator<>>;
}
```

(2) (since C++23)

1) The class template `std::generator` presents a view of the elements yielded by the evaluation of a coroutine.

2) Convenience alias template for the `generator` using the polymorphic allocator.

C++23 std::generator

Member functions

(constructor)	constructs a `generator` object (public member function)
(destructor)	effectively destroys the entire stack of yielded `generators` (public member function)
`operator=`	assigns a `generator` object (public member function)
`begin`	resumes the initially suspended coroutine and returns an iterator to its handle (public member function)
`end`	returns `std::default_sentinel` (public member function)

Inherited from `std::ranges::view_interface`

`empty`	returns whether the derived view is empty, provided only if it satisfies `sized_range` or `forward_range` (public member function of `std::ranges::view_interface<D>`)
`cbegin` (C++23)	returns a constant iterator to the beginning of the range (public member function of `std::ranges::view_interface<D>`)
`cend` (C++23)	returns a sentinel for the constant iterator of the range (public member function of `std::ranges::view_interface<D>`)
`operator bool`	returns whether the derived view is not empty, provided only if `ranges::empty` is applicable to it (public member function of `std::ranges::view_interface<D>`)

Nested classes

`promise_type`	the promise type (public member class)
`iterator`	the iterator type (exposition-only member class*)

Conclusion

- cooperative multitasking
- code conciseness
- lazy evaluation
- elegant iteration