C++ - parallelization and synchronization

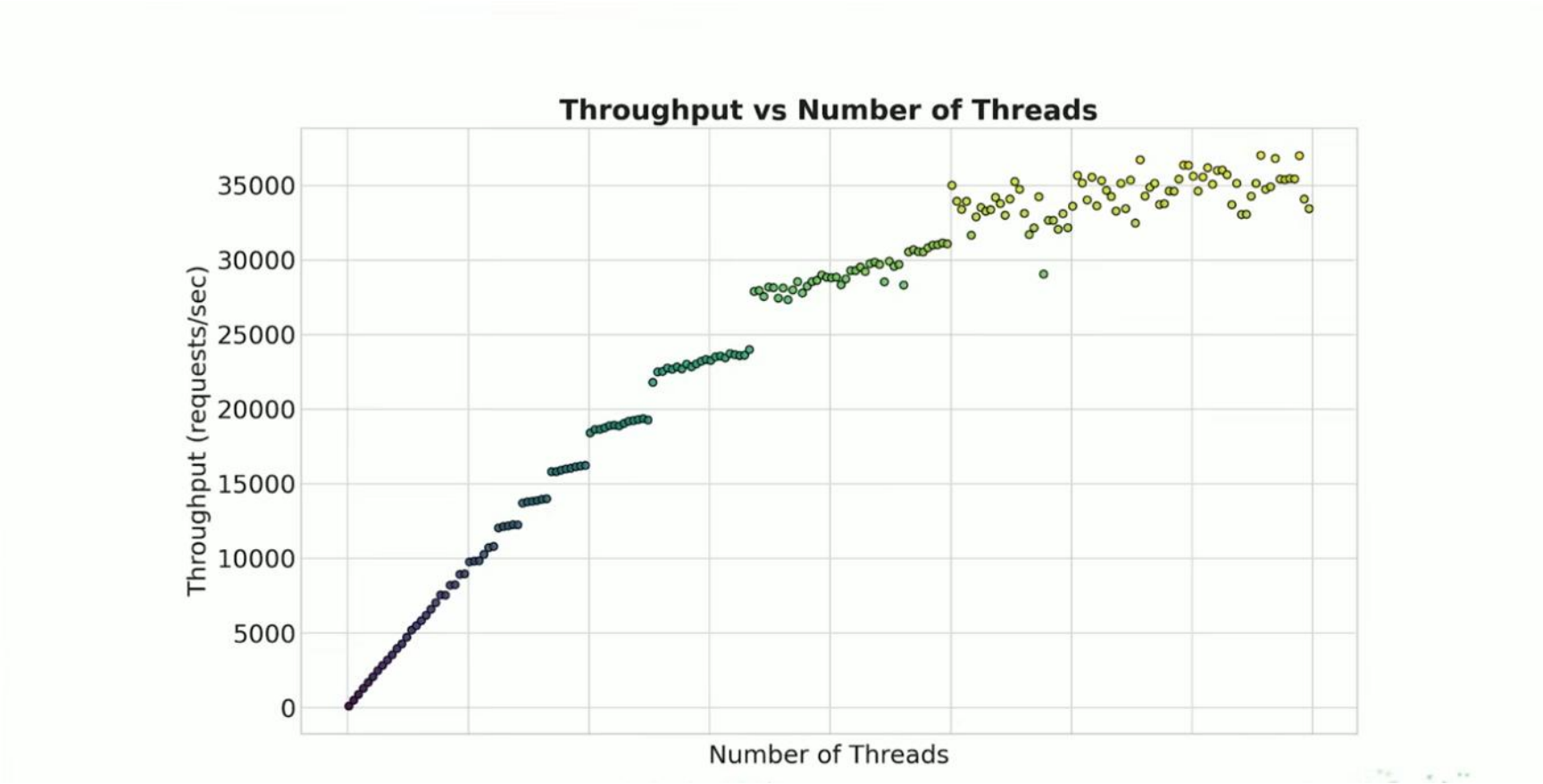
Coroutines

Tomáš Faltín

References

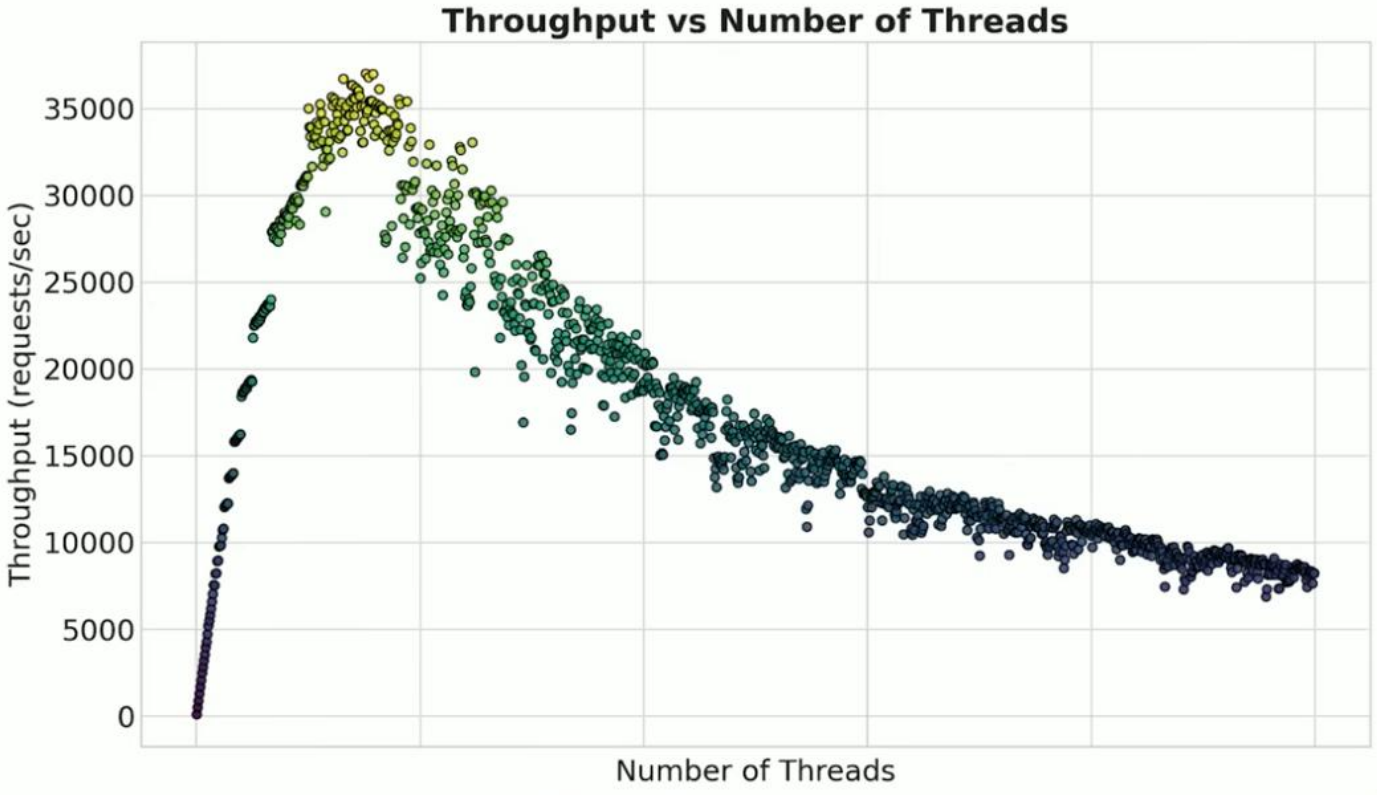
- https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html
- https://youtu.be/txffplpsSzg?si=HtVoibjwtu_vqAOE

Motivation



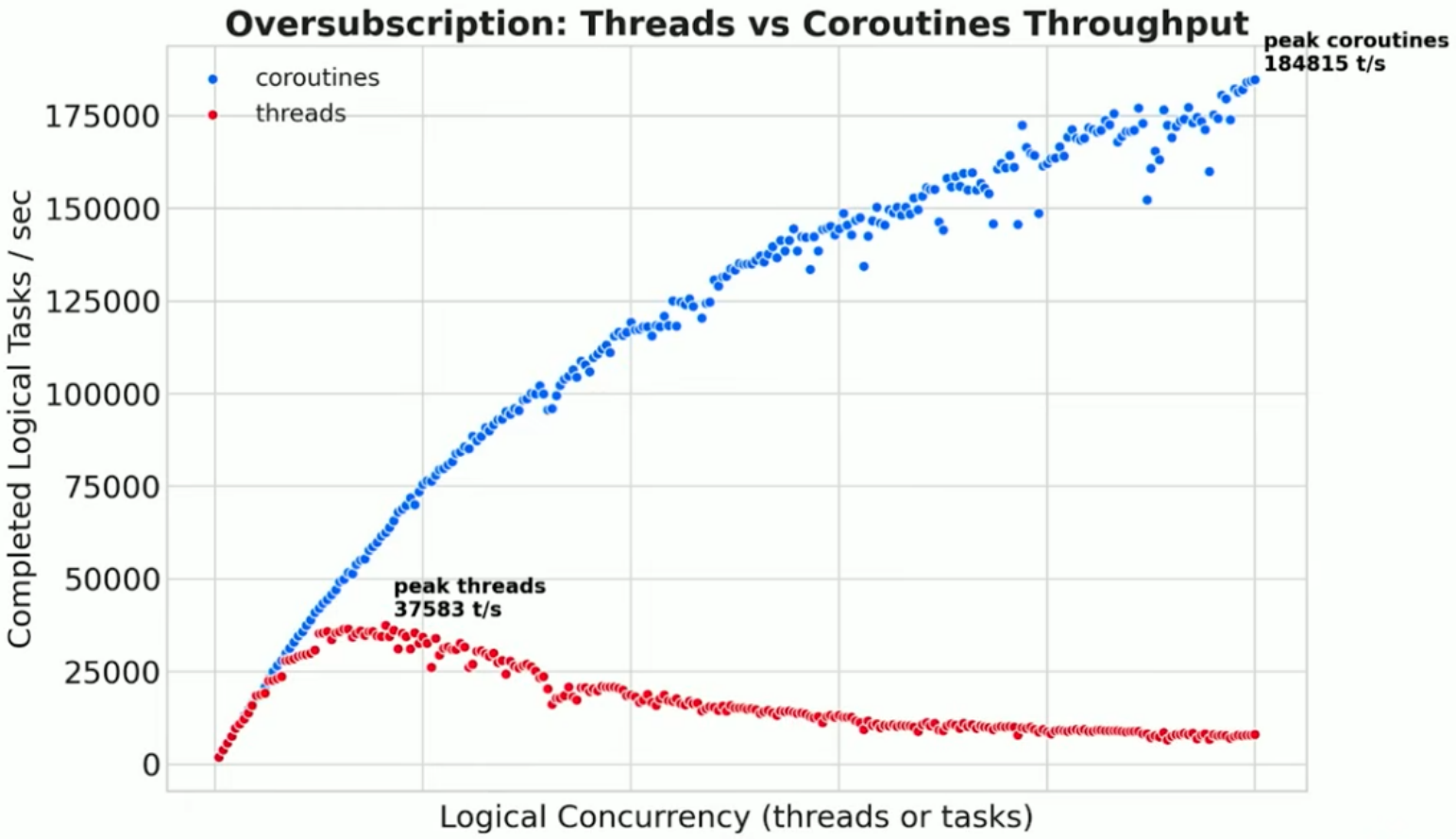
Source: https://youtu.be/txffplpsSzg?si=HtVoibjwtu_vqAOE

Motivation



Source: https://youtu.be/txffplpsSzg?si=HtVoibjwtu_vqAOE

Coroutines



Source: https://youtu.be/txffplpsSzg?si=HtVoibjwtu_vqAOE

What are coroutines?

- Like a subroutines
  - Can be called
  - Can return when completed
- But with some differences
  - Can suspend themselves
  - Can be resumed (by someone else)

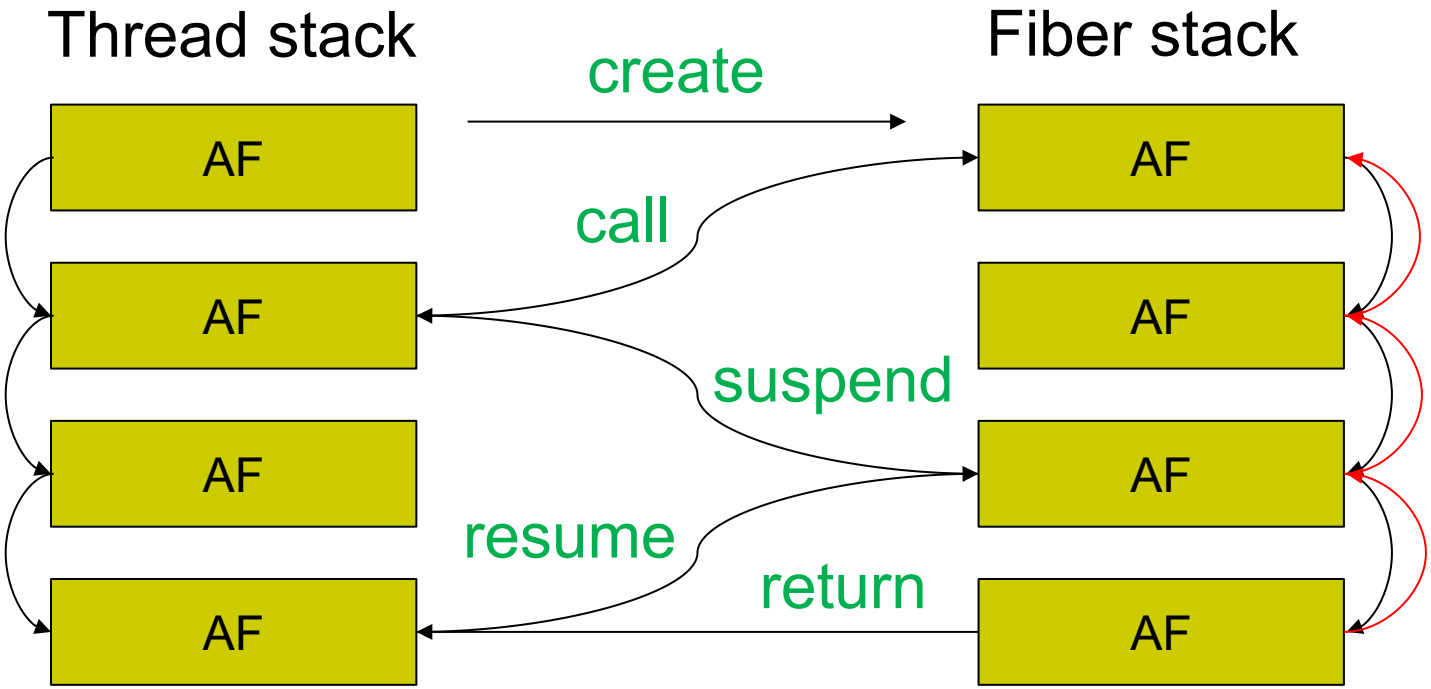
Why do we want coroutines?

- Cooperative multitasking
  - Cheaper context-switch compared with threads
- Event driven architectures
  - Asynchronous I/O
  - User interfaces
  - Simulations
- Generators
- Lazy evaluation

Stackful Coroutines

- Stackful
  - Fibers, green threads, etc.
  - They have their own call stack
  - Their lifetime is independent to the caller code
  - Can be attached and detached to/from threads
  - Cooperative scheduling
  - Can be implemented as a library, no need for language support

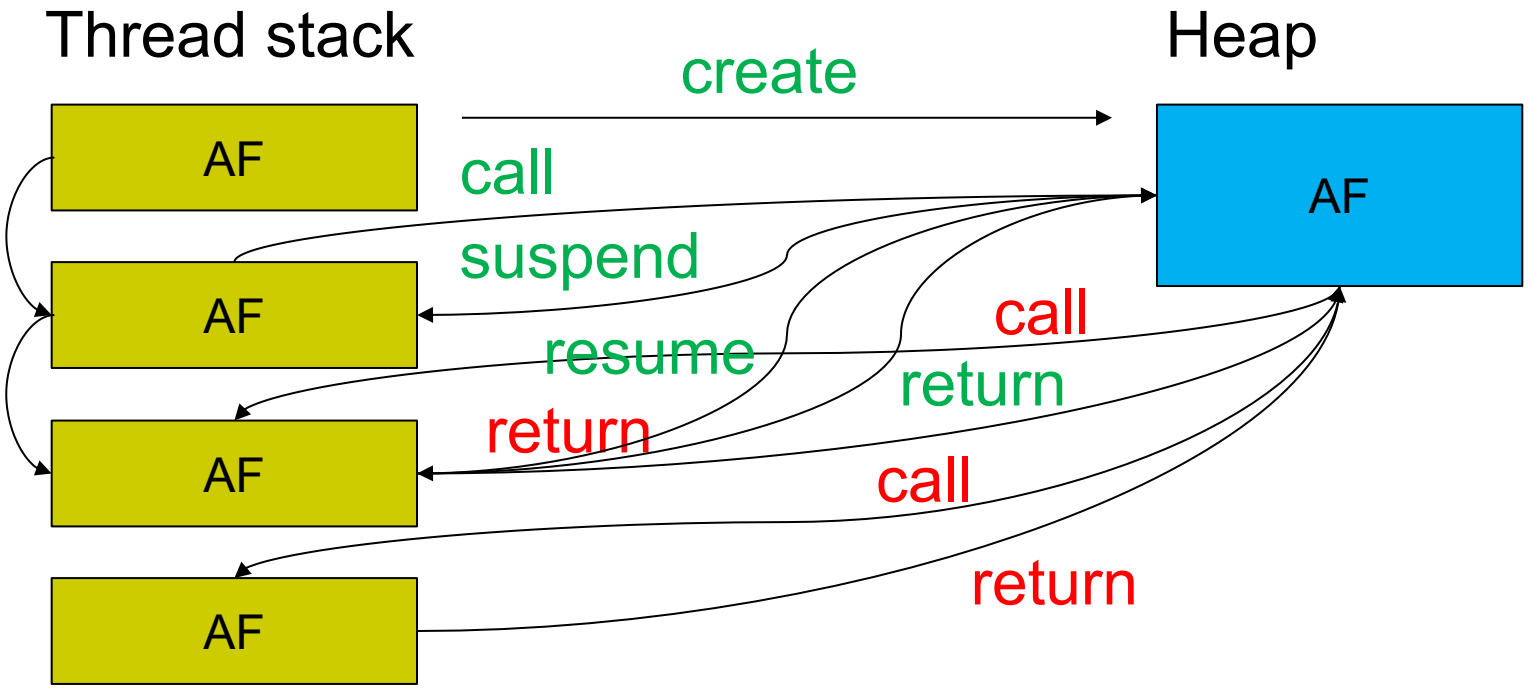
Stackful coroutines



Stackless coroutines

- Stackless
  - Use caller's stack
  - Can be suspended only from the top level function
    - All function calls made by coroutine must return before suspend
  - Coroutine state saved on the heap
  - Require language level support
  - Usually lighter
  - C++ 20

Stackless coroutines



C++ Coroutines

- How to detect a coroutine?
  - Any use of coroutine keyword transforms a function to the coroutine
    - Expressions `co_await`, `co_yield`
    - Statement `co_return`
- Restrictions
  - No functions with: variadic args, returns, auto
    - consteval, contexpr, ctors, dtors, …

co_wait – Example

```
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));
  }
}
```

Suspend execution until resumed

co_yield – Example

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

Suspend execution returning a value

co_return – Example

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

Complete execution returning a value

What does `co_await` do?

- All local variables in the current function are **saved to a heap** allocated object
- Creates a **callable object** that, when invoked, will resume execution of the coroutine at the point immediately following evaluation of the `co_await` expression
- Calls (jumps to) a method of `co_await`'s target object ***a***, passing that method the callable object from 2nd step

Coroutine Handles

- Like a C pointer
- Type `std::coroutine_handle<>`
- Call `coroutine_handle::destroy` to avoid leaking memory, it destroys the state
- Once destroyed, invoking coroutine handle has undefined behavior
- Coroutine handle is valid for the entire execution of a coroutine, even as control flows in and out of the coroutine

`co_await` again

```
co_await a;
```

- The compiler creates a coroutine handle and passes it to the method `a.await_suspend(coroutine_handle)`
- The type of ***a*** must support certain methods
  - Awaitable object or awaiter

co_await – Awaiter

```
struct Awaiter {
  std::coroutine_handle<> *hp_;
  constexpr bool await_ready() const noexcept { return false; }
  void await_suspend(std::coroutine_handle<> h) { *hp_ = h; }
  constexpr void await_resume() const noexcept
};

task counter(std::coroutine_handle<> *cont_out){
  Awaiter a{cont_out};
  for (unsigned i = 0;; ++i) {
    co_await a;
    std::cout << i << std::endl;
  }
};

int main(){
  std::coroutine_handle<> h;
  counter(&h);
  for(unsingned i = 0; i < 3; ++i) { h(); }
  h.destroy();
}
```

(1) Is it ready already?

(2) Called right after suspend

(3) **co_await expr**

`co_await` internals

```
auto res = co_await expr;


auto &&a = expr;
if(!a.await_ready()) {
  a.await_suspend(coroutine_handle);
  // suspension point
}
auto res = a.await_resume();
```

Predefined awaiters

- Include `<coroutine>`
  - `std::suspend_always`
    - `await_ready` returns false
  - `std::suspend_never`
    - `await_ready` returns true

Coroutine return object

- Coroutine return type R must be an object with nested type `R::promise_type`
  - Missing member function causes undefined behavior

```
struct task {
  struct promise_type {
    ReturnObject get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() { return {}; }
    void unhandled_exception() {}
  };
};
```

What does `co_yield` do?

- We need to get values from coroutines somehow
- `co_yield e;` is equivalent to `co_await p.yield_value(e);` where ***p*** is a promise

co_yield example – 1st part

```
struct task {
  struct promise_type {
    unsigned value_;

    task get_return_object() {
      return {
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() { return {}; }
    void unhandled_exception() {}
    std::suspend_always yield_value(unsigned value) {
      value_ = value;
      return {};
    }
  };

  std::coroutine_handle<promise_type> h_;
};
```

co_yield example – 2nd part

```
task counter() {
  for (unsigned i = 0;; ++i)
    // co yield i => co_await promise.yield_value(i)
    co_yield i;
}

int main(){
  auto h = counter().h_;
  auto &promise = h.promise();
  for (int i = 0; i < 3; ++i) {
    std::cout << "counter: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

What does `co_return` do?

- How to signal that the coroutine is complete?
  - Useful for finite streams
  - Coroutine can call `co_return e;` for returning a final value ***e***
    - Compiler inserts `p.return_value(e);`
  - Coroutine can call `co_return;` without value to end the coroutine without a final value
    - Compiler inserts `p.return_void();`
  - Coroutine execution falls off the end of the function
    - Equivalent to the previous case
- Check if coroutine is completed
  - You can call h.done()

co_return example – 1st part

```
struct task {
  struct promise_type {
    unsigned value_;

    ~promise_type() { /* do something */ }
    task get_return_object() {
      return {
        .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
      };
    }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_always final_suspend() { return {}; }
    void unhandled_exception() {}
    std::suspend_always yield_value(unsigned value) {
      value_ = value;
      return {};
    }
    void return_void() {}
  };

  std::coroutine_handle<promise_type> h_;
};
```

co_return example – 2nd part

```
task counter() {
  for (unsigned i = 0; i < 3; ++i)
    co_yield i;
  // falling off end of function or co_return;
}

int main() {
  auto h = counter().h_;
  auto &promise = h.promise();
  // Do NOT use while(h) (which checks h non-NULL)
  while (!h.done()) {
    std::cout << "counter: " << promise.value_ << std::endl;
    h();
  }
  h.destroy();
}
```

What about remaining member functions from promise?

- Compiler wraps coroutine function body

```
{
    promise-type promise promise-constructor-arguments ;
    try {
        co_await promise.initial_suspend() ;
        function-body
    } catch ( ... ) {
        if (!initial-await-resume-called)
            throw ;
        promise.unhandled_exception() ;
    }
final-suspend :
    co_await promise.final_suspend() ;
}
```

Automatic clean up

- Trick with `p.final_suspend()`
  - If `final_suspend` suspends the coroutine, the state remains valid and code outside of the routine is responsible for freeing the object by calling `destroy()`
  - If `final_suspend` does not suspend the coroutine, then the coroutine state will be automatically destroyed

Coroutines – Examples

```
struct promise;

struct coroutine : std::coroutine_handle<promise> {
  using promise_type = ::promise;
};

struct promise {
  coroutine get_return_object() {
    return {coroutine::from_promise(*this)};
  }
  std::suspend_always initial_suspend() noexcept { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }
  void return_void() {}
  void unhandled_exception() {}
};
```

Coroutines – Problems

```
struct S {
  int i;
  coroutine f(){
    std::cout << i;
    co_return;
  }
};

void bad(){
  coroutine h = S{0}.f();
  h.resume();
  h.destroy();
}
```

`S{0}` destroyed

Invokes `std::cout << i` after free

Coroutines – Problems

```
struct S {
  int i;
  coroutine f(){
    std::cout << i;
    co_return;
  }
};
void bad(){
  coroutine h = [i = 0]() -> coroutine {
    std::cout << i;
    co_return;
  }();
  h.resume();
  h.destroy();
}
```

lambda destroyed