An Introduction to the Coroutines TS

Toby Allsopp

WhereScape Software Limited

email: toby@mi6.gen.nz

twitter: @toby_allsopp

github: toby-allsopp

Introduction

Structure

- Context
- Basics
- Motivating examples
- Under the covers

What are we talking about?

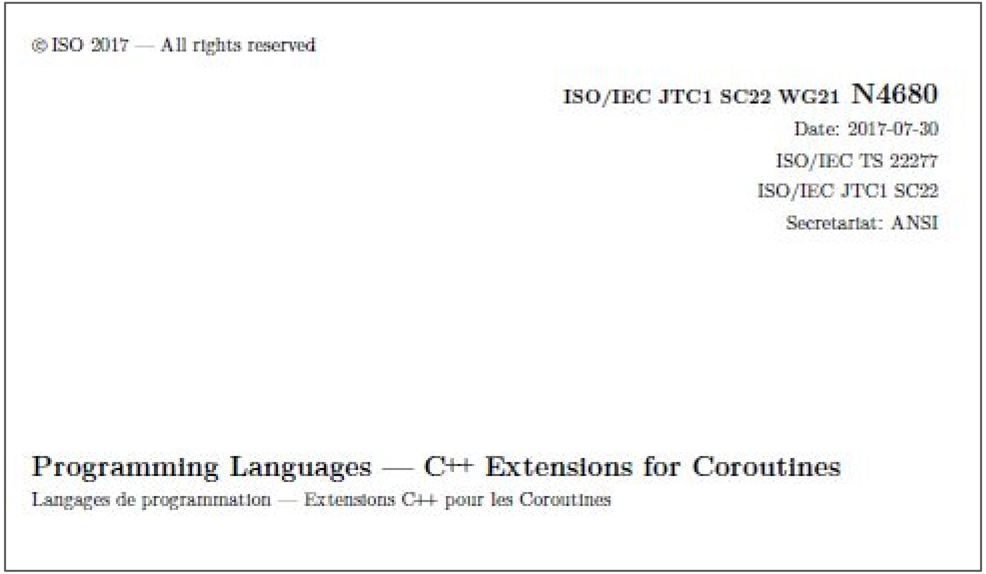
“Programming Languages — C++ Extensions for Coroutines”

aka N4680

a Technical Specification

henceforth: “the TS”

wg21.link/n4680



ISO/IEC JTC1 SC22 WG21 **N4680**

Date: 2017-07-30

ISO/IEC TS 22277

ISO/IEC JTC1 SC22

Secretariat: ANSI

Programming Languages — C++ Extensions for Coroutines

Langages de programmation — Extensions C++ pour les Coroutines

What’s *that* talking about?

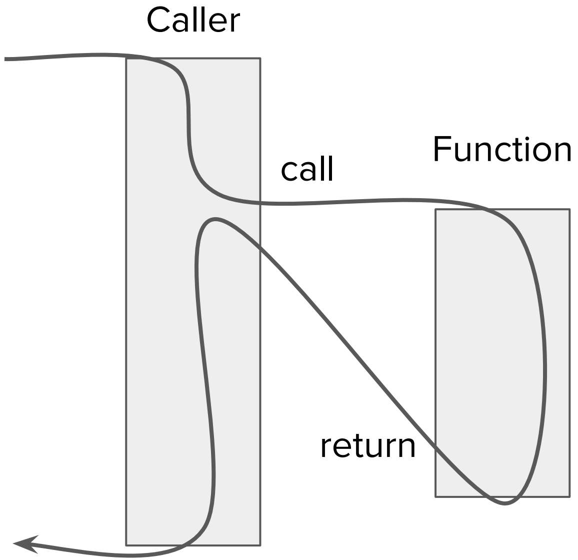
Coroutines, obviously!

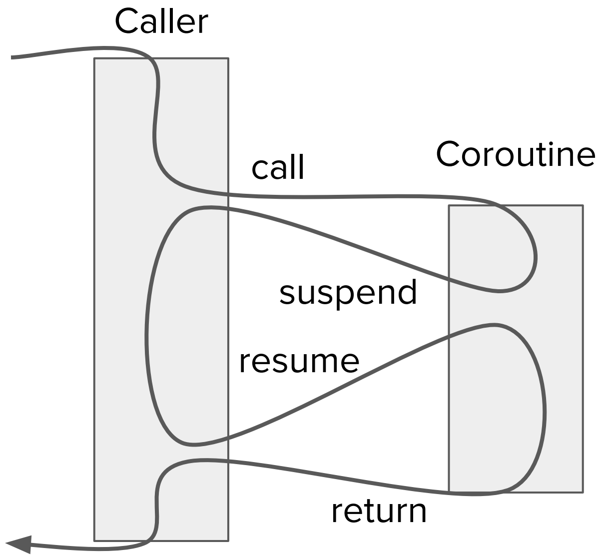
An extension to C++

Three new keywords and some new customization points

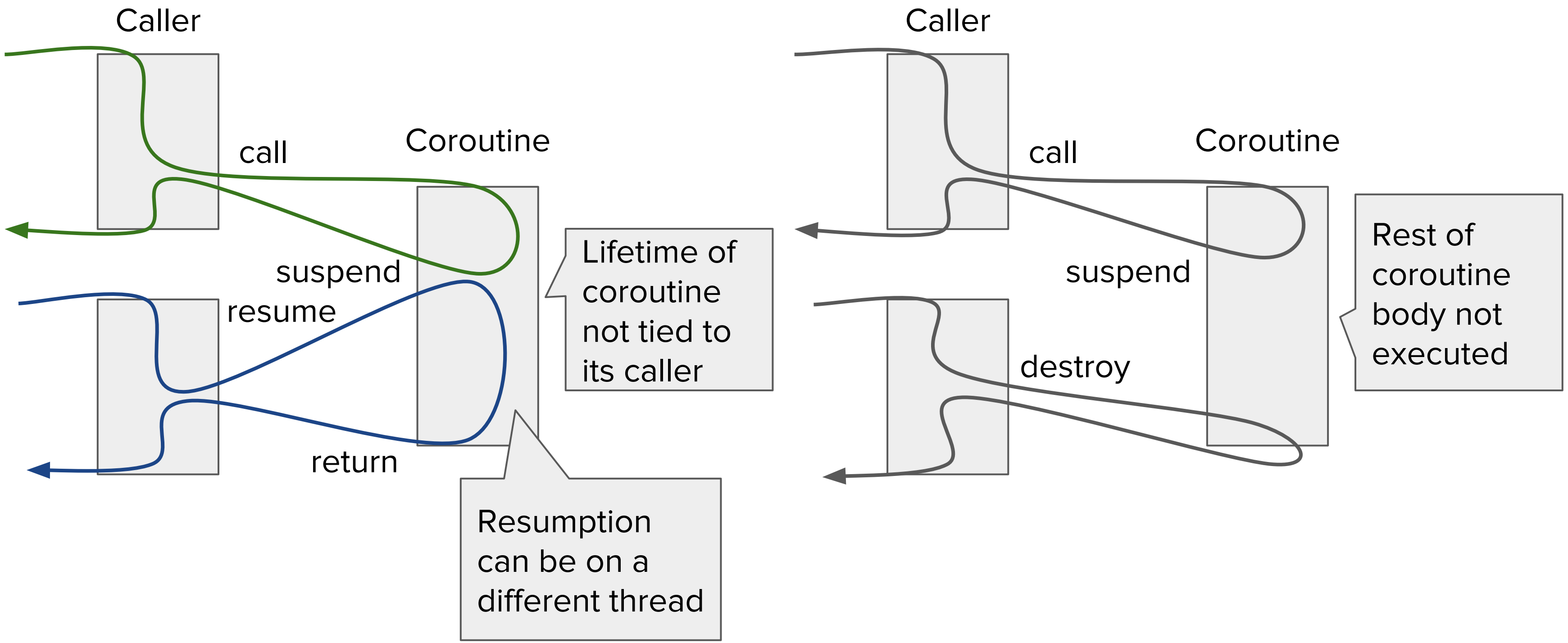
That enable some new (to C++) kinds of control flow

A coroutine is a generalization of a function





A coroutine is a generalization of a function



Motivating use-cases

Motivating use-cases

	Synchronous	Asynchronous
Single return value	Normal function	Task
Multiple return values	Generator	Async generator

Generators

A generator:

- computes a sequence of values
- one at a time
- on demand

Generators

Caller

```
bool caller() {
  for (int p : primes()) {
    use(p);
    if (p > 100) return true;
    if (finished(p)) break;
  }
  // many lines...
  return false;
}
```

Both caller and generator want to be in the driver's seat

Generator

```
vector<int> primes() {
  vector<int> ps;
  prime_tester tester;
  for (int i = 2;; ++i) {
    if (tester.is_prime(i))
      ps.push_back(i);
  }
  return ps; // unreachable
}
```

Clearly, this won't work!

Take 1 — callback

Caller

```
bool caller() {
  bool return_early = false;
  primes([&return_early](int p) {
    use(p);
    if (p > 100) {
      return_early = true;
      return true;  // finished
    }
    return finished(p);
  });
  if (return_early) return true;
  // many lines...
  return false;
}
```

Generator

```
template <class F>
void primes(F&& callback) {
  prime_tester tester;
  for (int i = 2;; ++i)
    if (tester.is_prime(i))
      if (callback(i)) return;
}
```

The generator is in the driver's seat

The caller is messed up

Take 2 — iterator

Caller

```
bool caller() {
  for (int p : primes()) {
    use(p);
    if (p > 100) return true;
    if (finished(p)) break;
  }
  // many lines...
  return false;
}
```

The caller is in the driver's seat

The generator is inside out

Generator

```
#include <range/v3/all.hpp>
using namespace ranges;

auto primes() {
  class range : public view_facade<range> {
    friend range_access;
    prime_tester tester_;
    struct cursor {
      range* range_;
      int i_ = 2;

      void next() {
        do {
          ++i_;
        } while (!range_->tester_.is_prime(i_));
      }
      bool done() const { return false; }
      int get() const { return i_; }
    };

    cursor begin_cursor() {
      return cursor{this, 2};
    }
  };
  return range{};
}
```

Take 3 — coroutine

Caller

```
bool caller() {
  for (int p : primes()) {
    use(p);
    if (p > 100) return true;
    if (finished(p)) break;
  }
  // many lines...
  return false;
}
```

Both caller and generator are in the driver's seat

Generator

```
generator<int> primes() {
  prime_tester tester;
  for (int i = 2;; ++i) {
    if (tester.is_prime(i))
      co_yield i;
  }
}
```

`generator<T>` is a class that uses a coroutine to compute values

Async

An asynchronous function:

- returns promptly to its caller (does not block)
- returns an object representing the outstanding work to be done
- provides a way to get the eventual result

(Not) async

- Imagine some API for connecting to and reading from a socket
- It's pretty easy to use if it's blocking

```
class Socket {
 public:
  template <size_t N>
  int read(array<char, N>& buf);
};

Socket connect(string url);
```

```
string download(string url) {
  auto s = connect(url);
  string result;
  array<char, 1024> buf;
  while (int n = s.read(buf)) {
    result.append(buf.data(), n);
  }
  return result;
}
```

(Still not) async

Now imagine the API is non-blocking

```
class Socket {
 public:
  template <size_t N>
  auto read(array<char, N>& buf)
    -> Task<int>;
};

Task<Socket> connect(string url);
```

Task is a fictional class template representing a value that may not be available yet

```
string download(string url) {
  auto s = connect(url).get();
  string result;
  array<char, 1024> buf;
  while (int n = s.read(buf).get()) {
    result.append(buf.data(), n);
  }
  return result;
}
```

`Task::get()` blocks until the value is available

Async (with `.then`)

```
struct State {
  explicit State(Socket socket)
    : socket(move(socket)) {}
  Socket socket;
  array<char, 1024> buffer;
  string result;
};
```

```
Task<string> download(string url) {
  return connect(move(url)).then([](auto socket) {
    auto state = make_shared<State>(move(socket));
    return while_task([state]() {
             return state->socket.read(state->buffer).then([state](int n) {
               if (n == 0) return make_ready_task(false);
               state->result.append(state->buffer.data(), n);
               return make_ready_task(true);
             });
           })
        .then([state]() { return make_ready_task(state->result); });
  });
}
```

Callbacks turn the logic inside out

`Task::then()` returns a new task that runs a *continuation* when the value is available

Remember what we were trying to do?

```
    string  download(string url) {
  auto s =           connect(url);
  string result;
  array<char, 1024> buf;
  while (int n =           s.read(buf)) {
    result.append(buf.data(), n);
  }
    return result;
}
```

Make this asynchronous

It’s easy with coroutines!

```
Task<string> download(string url) {
  auto s = co_await connect(url);
  string result;
  array<char, 1024> buf;
  while (int n = co_await s.read(buf)) {
    result.append(buf.data(), n);
  }
  co_return result;
}
```

Requires some work in Task to be usable as a coroutine return type

Async generators

An asynchronous generator:

- computes a sequence of values
- one at a time
- on demand
- asynchronously

Async primes

```
Task<bool> caller()
{
  for co_await (int p : primes)
  {
    use(p);
    if (p > 100) co_return true;
    if (finished(p)) break;
  }
  // many lines...
  co_return false;
}
```

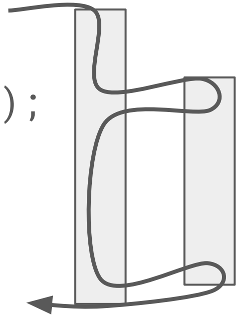
```
AsyncGenerator<int> primes()
{
  async_prime_tester tester;
  for (int i = 2;; ++i)
  {
    if (co_await
          tester.is_prime(i))
      co_yield i;
  }
}
```

`AsyncGenerator`s can use both `co_await` and `co_yield`

Coroutine transformations and customization points

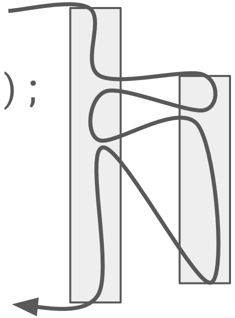
Very simple coroutine

```
void caller() {
  auto lazy_result = calculate();


}
```



```
lazy<int> calculate() {
  int result = hard_work();
  co_return result;
}
```

```
int caller() {
  auto lazy_result = calculate();
  return lazy_result.get();

}
```



```
lazy<int> calculate() {
  int result = hard_work();
  co_return result;
}
```

Coroutine body transformation

```
lazy<int> calculate()

{



   int result = hard_work();
   co_return result;






}
```

the promise is actually stored in the coroutine state

local variables may actually be stored in the coroutine state

```
using promise_type =
    coroutine_traits<lazy<int>>::promise_type;
{
  promise_type p;
  auto r = p.get_return_object();
  co_await p.initial_suspend();
  try {
    int result = hard_work();
    p.return_value(result);
    goto final_suspend;
  } catch (...) {
    p.unhandled_exception();
  }
final_suspend:
  co_await p.final_suspend();
destroy: ;
}
```

the return object is really like a local variable

Coroutine state

stores promise, parameter copies, local variables and any information needed to resume the coroutine at the correct point

dynamically allocated (can be elided in certain circumstances)

created when the coroutine is called

destroyed when control flows off the end of the transformed coroutine body

promise
parameter copies
local variables
resume point

`co_await` transformation

All coroutines `co_await` at least once, on `p.initial_suspend()`

```
auto y = co_await x;
```

```
auto h = coroutine_handle<P>::from_promise(p);
auto a = p.await_transform(x); // optional
auto e = operator co_await(a); // optional
if (!e.await_ready()) {
  // suspended
  e.await_suspend(h);
  if (/* still suspended */)
    // return to caller
resume: ;
  }
  // resumed
}
auto y = e.await_resume();
```

`co_yield` transformation

`co_yield` is almost the same as `co_await`

```
auto y = co_yield x;
```

```
auto h = coroutine_handle<P>::from_promise(p);
auto a = p.yield_value(x);
auto e = operator co_await(a); // optional
if (!e.await_ready()) {
  // suspended
  e.await_suspend(h);
  if (/* still suspended */)
    // return to caller
resume: ;
  }
  // resumed
}
auto y = e.await_resume();
```

`for co_await` transformation

```
for co_await (auto x : xs) {




  use(x);
}
```

```
{
  auto __end = xs.end();
  for (auto __begin = co_await xs.begin();
       __begin != __end;
       co_await ++__begin) {
    auto x = *__begin;
    use(x);
  }
}
```

Coroutine handle

non-owning; destructor doesn't do anything

might not refer to a coroutine at all

provides access to the promise

can get one from the promise

only way to resume or destroy a coroutine

```
template <typename Promise>
struct coroutine_handle {
  constexpr explicit
    operator bool() const noexcept;

  Promise& promise() const;

  static coroutine_handle
    from_promise(Promise&);

  void resume();
  void destroy();
};
```

Return values

Whenever a coroutine suspends or returns, it returns control to its caller (the first time) or its resumer (subsequent times).

```
lazy<int> v = calculate(); // caller
```

When returning to its caller, the return object is used - it is converted to the return type of the coroutine.

```
h.resume(); // resumer
h.destroy(); // resumer
```

When returning to a resumer there is no return value.

Implementation for `lazy` 1/n

```
namespace stdx = std::experimental;

template <typename T>
struct lazy_promise;

template <typename T>
class lazy {
 private:
    using handle_type =
          stdx::coroutine_handle<
            lazy_promise<T>>;

    lazy(lazy_promise<T>& p)
        : h(handle_type::
                from_promise(p)) {}

    handle_type h;

    friend lazy_promise<T>;
```

for brevity; don't do this in a header file!

`lazy` and `lazy_promise` refer to each other

to control the coroutine

Implementation for `lazy` 2/n

```cpp
public:
  T get() {
    if (!h.promise().value) {
      h.resume();
    }
    return *h.promise().value;
  }

  ~lazy() {
    if (h) h.destroy();
  }
};
```

executes remainder of coroutine synchronously

also need copy/move constructor and assignment

```cpp
template <typename T>
struct lazy_promise {
  std::optional<T> value;

  auto get_return_object() {
    return lazy<T>(*this); }
  auto initial_suspend() {
    return stdx::suspend_always{}; }
  void return_value(T x) {
    value = std::move(x); }
  void unhandled_exception() {}
  auto final_suspend() {
    return stdx::suspend_always{}; }
};
```

to store the lazily-computed value

Implementation for `lazy` 3/3

```
namespace std::experimental {
  template <typename T, typename... Args>
  struct coroutine_traits<lazy<T>, Args...> {
    using promise_type = lazy_promise<T>;
  };
}
```

connects the signature of the coroutine to the promise type

I want to use this stuff now!

Working coroutine implementations:

- Visual Studio 2017 (`/await`)
- clang 5.0 with libc++ 5.0 (`-fcoroutines-ts -stdlib=libc++`)

Coroutine abstraction libraries:

- cppcoro (https://github.com/lewissbaker/cppcoro)
  - task, generator, async_generator, async_mutex, ...
- range-v3 (https://github.com/ericniebler/range-v3)
  - generator
- others? Let me know!

Summary

Coroutines are a generalization of functions that allow for many new and interesting control flows.

Usual suspects:

- generators
- async
- async generators

They require support from library types in order to define what the syntax means.

Thank you!

Some unexpected things you can do

Some operations returning `optional`

```
optional<string> read_word(istream& s);
optional<int> parse_int(istream& s);
optional<double> parse_double(istream& s);

// “3 1.2 3.4 5.6” -> vector{1.2, 3.4, 5.6}
optional<vector<double>> parse_vector(istream& s);
```

Composing operations returning `optional`

```
optional<vector<double>> parse_vector(istream& s) {
  optional<int> n = parse_int(s);
  if (!n) return {};
  vector<double> result;
  for (int i = 0; i < *n; ++i) {
    optional<double> x = parse_double(s);
    if (!x) return {};
    result.push_back(*x);
  }
  return result;
}
```

Composing operations in a coroutine

```
optional<vector<double>> parse_vector(istream& s) {
  int n = co_await parse_int(s);

  vector<double> result;
  for (int i = 0; i < n; ++i) {


    result.push_back(co_await parse_double(s));
  }
  co_return result;
}
```

How can this be?

need to specialize `coroutine_traits` for functions returning `optional`

```
namespace std::experimental {
  template <typename T, typename... Args>
  struct coroutine_traits<optional<T>, Args...> { // UB!
    using promise_type = optional_promise<T>;
  };
}
```

- this is not allowed if we're talking about std::optional - there needs to be a user-defined type in there somewhere
- but it works anyway

How does it work?

It’s all in the awaitable:

```
template <typename T>
struct optional_promise {
  template <typename U>
  auto await_transform(optional<U> e) {
    return optional_awaitable<U>{move(e)};
  }
};
```

It’s all in the awaitable

```
template <typename U>
struct optional_awaitable {
  optional<U> o;

  auto await_ready() { return o.has_value(); }
  auto await_resume() { return o.value(); }

  template <typename T>
  void await_suspend(coroutine_handle<maybe_promise<T>> h) {
    h.promise().data->emplace(nullopt);
    h.destroy();
  }
};
```

suspend if the optional does not contain a value

if it does contain a value, return that value

if no value, destroy the coroutine

But I want more information about what failed

Luckily for you, the exact same approach works with types like `expected`.

```
enum class ParseError { EndOfStream, BadFormat };

template <typename Value>
using parse_result = expected<Value, ParseError>;

parse_result<string> read_word(istream& s);
parse_result<int> parse_int(istream& s);
parse_result<double> parse_double(istream& s);
parse_result<vector<double>> parse_vector(istream& s);
```

I want to play with `optional` coroutines

- https://github.com/toby-allsopp/coroutine_monad
  - works with `std::optional`
  - but only on clang
  - has a basic `expected` implementation
- `folly::Optional` (https://github.com/facebook/folly)
  - not `std::optional`
  - works on clang *and* MSVC
  - would be pretty easy to get `folly::Expected` working as well

Type-erased callable

- This idea is from Gor Nishanov
- The observation is that the coroutine body can hold onto more type information than is reflected in the return type of the coroutine

Type-erased callable

```
template <typename Ret,
          typename... Args>
class func {
  template <typename F>
  static func create(F f) {
    co_yield f;
  }

  Ret operator()(Args… args) {
    h.promise().args = {args…};
    h.resume();
    return h.promise().ret;
  }
};
```

function type erased here

function type recovered here

```
template <typename Ret,
          typename... Args>
struct func_promise() {
  tuple<Args…> args;
  Ret ret;

  template <typename F>
  void yield_value(F f) {
    ret = apply(f, args);
  }
};
```

Type-erased callable

- If the compiler can see what you put in and where you call it then all of the coroutine machinery can be optimized away
- Otherwise, overhead is probably similar to `std::function`
- An intriguing technique
- Some code to start playing with: https://github.com/toby-allsopp/coroutine_func