Introduction to the Coroutines TS

Part I

Toby Allsopp
toby@mi6.gen.nz

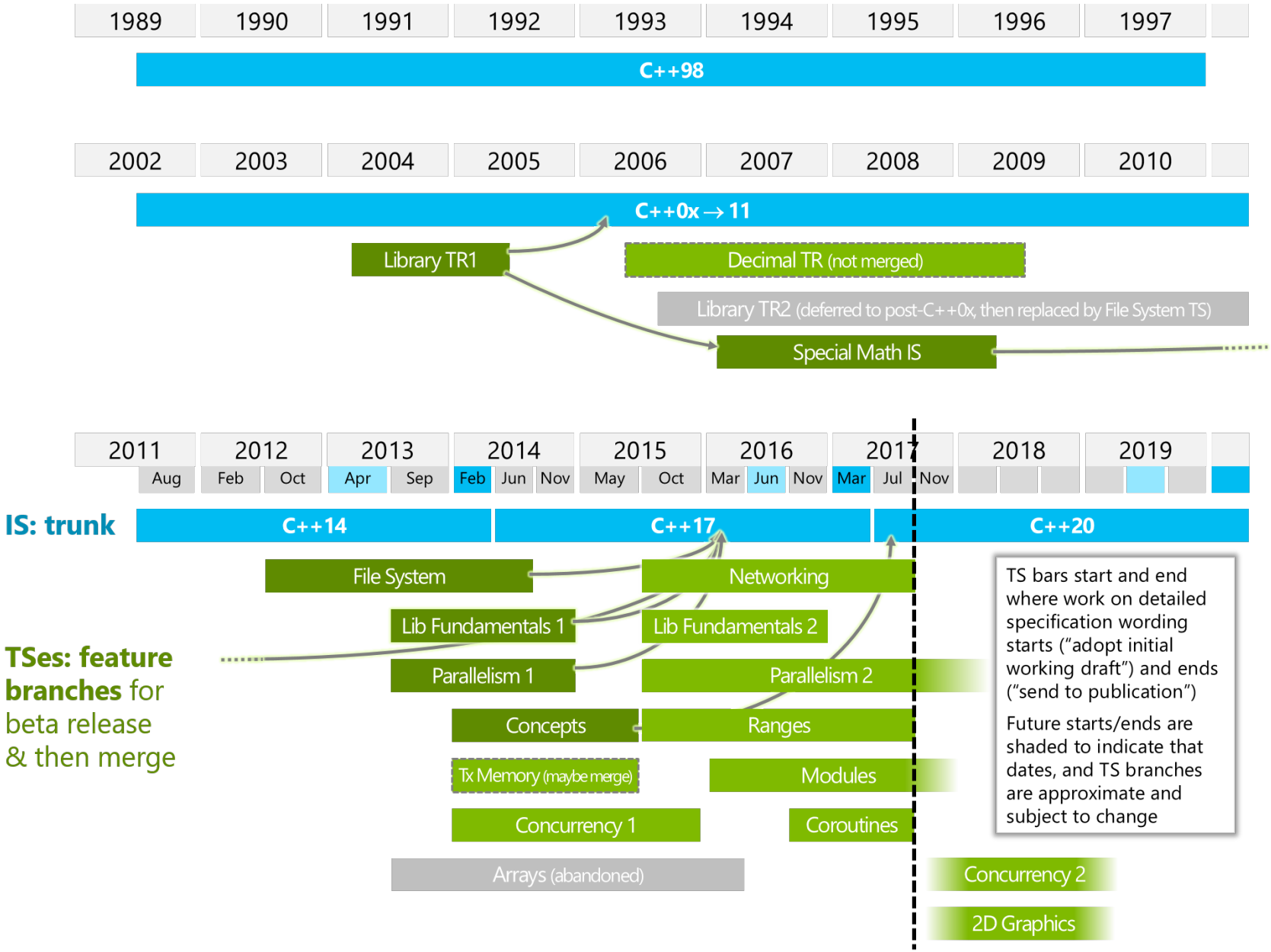
Auckland C++ Meetup
9 August 2017

Overview

- The Technical Specification
- What’s a Coroutine?
- What are they good for?
- How can I use this stuff today?

The Technical Specification

What's a technical specification?



The coroutines technical specification

- *Programming Languages — C++ Extensions for Coroutines*
- TS “draft”: `http://wg21.link/n4680` (published 2017-07-30)
- Championed by Gor Nishanov (MSFT)
- Voted for publication at the July ISO C++ committee meeting
  - All national body comments have been addressed
  - No guarantee it will be included in C++20

The coroutines technical specification

- *Programming Languages — C++ Extensions for Coroutines*
- TS “draft”: `http://wg21.link/n4680` (published 2017-07-30)
- Championed by Gor Nishanov (MSFT)
- Voted for publication at the July ISO C++ committee meeting
  - All national body comments have been addressed
  - No guarantee it will be included in C++20
- Implemented in MSVC and Clang
  - Visual Studio 2015 (minor differences wrt. TS)
  - Clang trunk (will be version 5.0 I think)

What’s a Coroutine?

What are these things?

- Lots of subtly different things in many different languages are called “coroutines”
- It’s important not to get confused by experience with other kinds of coroutines

What are these things?

- Lots of subtly different things in many different languages are called “coroutines”
- It’s important not to get confused by experience with other kinds of coroutines
- Generalization of functions — coroutines *are* functions
- Can be suspended and resumed

What are they good for?

Canonical coroutine applications

- Generators
- Async tasks
- Async generators

Example: printing primes

Here's a utility class we'll use on future slides; nothing to do with coroutines yet.

```
1  struct prime_tester {
2    vector<int> primes;
3
4    bool test(int n) {
5      if (none_of(primes.begin(), primes.end(),
6              [n](int p) { return n%p == 0; })) {
7        primes.push_back(n);
8        return true;
9      }
10     return false;
11   }
12 };
```

Example: printing primes

Callback (push) version:

```
1 template <typename Print>
2 void print_primes(Print print) {
3   prime_tester tester;
4   for (int n = 2; ; ++n)
5     if (tester.test(n))
6       if (!print(n))
7         break;
8 }
```

Primes generator

Generator (pull) version:

```
1
2  generator<int> primes() {
3    prime_tester tester;
4    for (int n = 2; ; ++n)
5      if (tester.test(n))
6        co_yield n;
7
8  }
```

Primes generator

To use the callback version:

```
print_primes([](int n) {
  cout << n << "\n";
  return n < 100;
});
```

Primes generator

To use the callback version:

```
print_primes([](int n) {
  cout << n << "\n";
  return n < 100;
});
```

To use the generator, just iterate over it:

```
for (int n : primes()) {
  cout << n << "\n";
  if (n >= 100) break;
}
```

Primes generator

```
1 generator<int> primes() {
2   prime_tester tester;
3   for (int n = 2; ; ++n)
4     if (tester.test(n))
5       co_yield n;
6   }
7 }
```

- **co_yield** is the first of three new keywords
- `generator<T>` is a class with `begin()` and `end()` members
- `generator` is not provided by the TS but you can make it yourself

Example: reading from a socket

Synchronously:

```
1     string  connect_and_read(string host,
2                              int port) {
3   string result;
4   auto socket =           connect(host, port);
5   array<char, 1024> buffer;
6   while (int nread =
7                       socket.read(buffer))
8     result.append(buffer, nread);
9     return result;
10 }
```

Example: reading from a socket

Asynchronously:

```
1  task<string> connect_and_read(string host,
2                                int port) {
3    string result;
4    auto socket = co_await connect(host, port);
5    array<char, 1024> buffer;
6    while (int nread =
7              co_await socket.read(buffer))
8      result.append(buffer, nread);
9    co_return result;
10 }
```

- Two more new keywords: **co_await** and **co_return**
- `task` could be `std::future` or any other class that has the coroutine customization points implemented

Example: asynchronous prime stream

```
struct async_pt {
  static task<async_pt> create();
  task<bool> test(int n);
};
```

```
async_generator<int> primes() {
  auto tester = co_await async_pt::create();
  for (int n = 2; ; ++n)
    if (co_await tester.test(n))
      co_yield n;
}
```

```
for co_await (int n : primes()) {
  cout << n << "\n";
  if (n >= 100) break;
}
```

Example: asynchronous prime stream

```
struct async_pt {
  socket_t socket;
  static task<async_pt> create() {
    auto s = co_await connect("host", 1234);
    co_return async_pt{s};
  }
  task<bool> test(int n) {
    co_await socket.write(to_string(n) + "\n");
    byte b = co_await socket.read();
    co_return b != 0;
  }
};
```

Example: asynchronous prime stream

```
async_generator<int> primes() {
  auto tester = co_await async_pt::create();
  for (int n : ints(2))
    if (co_await tester.test(n))
      co_yield n;
}
```

- `async_generator<T>` is a hypothetical class representing an asynchronous stream
- can use both **`co_await`** and **`co_yield`**

Example: asynchronous prime stream

The interface of `async_generator` is a lot like `generator` but `begin()` and **`operator`**`++()` are asynchronous.

```
1  template <typename T>
2  class async_generator {
3    task<iterator> begin();
4    iterator end();
5
6    class iterator {
7      task<void> operator++();
8      T& operator*() const;
9    };
10 };
```

Example: asynchronous prime stream

What about this **`for co_await`** thing?

```
1 for co_await (int n : primes()) {
2   cout << n << "\n";
3   if (n >= 100) break;
4 }
```

Example: asynchronous prime stream

What about this **`for co_await`** thing?

```
for co_await (int n : primes()) {
  cout << n << "\n";
  if (n >= 100) break;
}
```

It's equivalent to:

```
auto&& ag = primes();
for (auto it = co_await ag.begin();
     it != ag.end();
     co_await ++it) {
  int n = *it;
  cout << n << "\n";
  if (n >= 100) break;
}
```

How can I use this stuff today?

Using coroutines today

- Compilers and standard libraries
  - MSVC from Visual Studio 2015 Update 3 (`/await`)
  - Clang with libc++ recent trunk builds (`-stdlib=libc++ -fcoroutines-ts`)
- Supporting libraries
  - cppcoro (`https://github.com/lewissbaker/cppcoro`)
  - range-v3 (`https://github.com/ericniebler/range-v3`)